



# Intelligent Control Station

## *i*<sup>3</sup> IEC-61131 Tutorial



Think **inside** the box

## Contents

<b>TABLE OF CONTENTS .....</b>	<b>1</b>
<b>SAFETY GUIDELINES.....</b>	<b>3</b>
<i>Safety Warnings and Guidelines .....</i>	<i>3</i>
<i>Grounding .....</i>	<i>4</i>
IEC 61131 LOGIC .....	5
<i>IEC Users Guide .....</i>	<i>5</i>
<i>Languages .....</i>	<i>96</i>
<i>Function Blocks .....</i>	<i>149</i>
LOGIC MODULES.....	428
<i>IEC Modules .....</i>	<i>429</i>
RECIPES .....	442
<i>Overview .....</i>	<i>442</i>
<i>Creating a Recipe .....</i>	<i>443</i>
<i>Recipe Editor .....</i>	<i>443</i>
<i>Recipe Editor .....</i>	<i>444</i>
<i>Editing Ingredient Properties .....</i>	<i>445</i>
<i>Editing Recipe Data.....</i>	<i>447</i>
<i>Renaming Products.....</i>	<i>447</i>
<i>Auto Allocate Ingredient Register.....</i>	<i>447</i>
<i>Editing Current Recipe .....</i>	<i>448</i>
<i>Configuring Product Register.....</i>	<i>448</i>
USING SETPOINTS .....	450
<i>Editing the Setpoints .....</i>	<i>452</i>
<i>Formatting Setpoints.....</i>	<i>452</i>
<i>Transferring and Verifying Values with the Controller.....</i>	<i>453</i>
<i>Setpoint Tables, Setpoint Values, Uploading and Downloading .....</i>	<i>453</i>
<i>Printing the Setpoints.....</i>	<i>454</i>
HOW TO CHECK A PROGRAM FOR ERRORS.....	455
<i>Error and Warning List .....</i>	<i>456</i>
PRINT SETUP DIALOG .....	457
CLEARING THE CONTROLLER MEMORY .....	459

Information in this document is subject to change without notice and does not represent a commitment on the part of IMO Precision Controls.

# SAFETY GUIDELINES

## Safety Warnings and Guidelines

When found on the product, the following symbols specify:



**Warning:** Consult user documentation.



**Warning:** Electrical Shock Hazard.

**WARNING – EXPLOSION HAZARD – Do not disconnect equipment unless power has been switched off or the area is known to be non-hazardous**

**WARNING:** To avoid the risk of electric shock or burns, always connect the safety (or earth) ground before making any other connections.

**WARNING:** To reduce the risk of fire, electrical shock, or physical injury it is strongly recommended to fuse the voltage measurement inputs. Be sure to locate fuses as close to the source as possible.

**WARNING:** Replace fuse with the same type and rating to provide protection against risk of fire and shock hazards.

**WARNING:** In the event of repeated failure, do not replace the fuse again as a repeated failure indicates a defective condition that will not clear by replacing the fuse.

**WARNING – EXPLOSION HAZARD – Substitution of components may impair suitability for Class I, Division 2**

**WARNING - The USB parts are for operational maintenance only. Do not leave permanently connected unless area is known to be non-hazardous**

**WARNING – EXPLOSION HAZARD - BATTERIES MUST ONLY BE CHANGED IN AN AREA KNOWN TO BE NON-HAZARDOUS**

**WARNING - Battery May Explode If Mistreated. Do Not Recharge, Disassemble or Dispose of in Fire**

**WARNING:** Only qualified electrical personnel familiar with the construction and operation of this equipment and the hazards involved should install, adjust, operate, or service this equipment. Read and understand this manual and other applicable manuals in their entirety before proceeding. Failure to observe this precaution could result in severe bodily injury or loss of life.

- a. All applicable codes and standards need to be followed in the installation of this product.

- b. For I/O wiring (discrete), use the following wire type or equivalent: Belden 9918, 18 AWG or larger.

Adhere to the following safety precautions whenever any type of connection is made to the module.

- a. Connect the green safety (earth) ground first before making any other connections.
- b. When connecting to electric circuits or pulse-initiating equipment, open their related breakers. Do not make connections to live power lines.
- c. Make connections to the module first; then connect to the circuit to be monitored.
- d. Route power wires in a safe manner in accordance with good practice and local codes.
- e. Wear proper personal protective equipment including safety glasses and insulated gloves when making connections to power circuits.
- f. Ensure hands, shoes, and floor is dry before making any connection to a power line.
- g. Make sure the unit is turned OFF before making connection to terminals. Make sure all circuits are de-energized before making connections.
- h. Before each use, inspect all cables for breaks or cracks in the insulation. Replace immediately if defective.

## **Grounding**

Grounding is covered in various chapters within this manual.

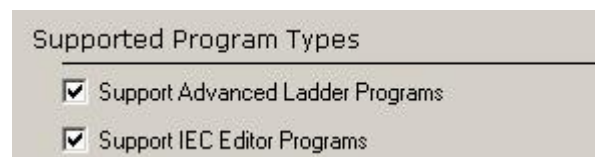
# IEC 61131 Logic

## IEC Users Guide

### IEC 61131-3 Programming Environment

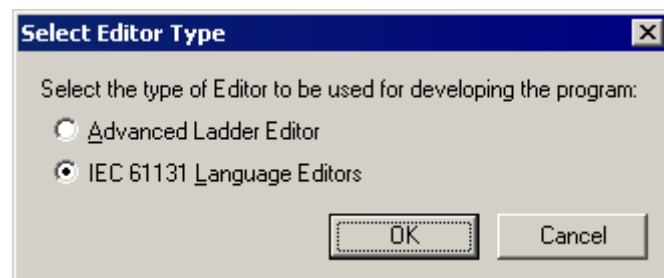
**i<sup>3</sup> Configurator** v9.0 supports integrated editing environment for working with IEC programs. This feature allows creating, editing and debugging IEC programs within **i<sup>3</sup> Configurator** main window.

A new configuration option 'Supported Program Types' has been added under Tools | Application Settings. This option allow users to create new programs in Advanced Ladder or IEC Editor or both.



If only 'Support IEC Editor Programs' option is checked any new program created will automatically be an IEC program.

If both 'Support Advanced Ladder Programs' and 'Support IEC Editor Programs' option is checked the type of any new program created will be configurable by the user.



In each case, opening an existing program will open it in its original mode with the appropriate editors enabled.

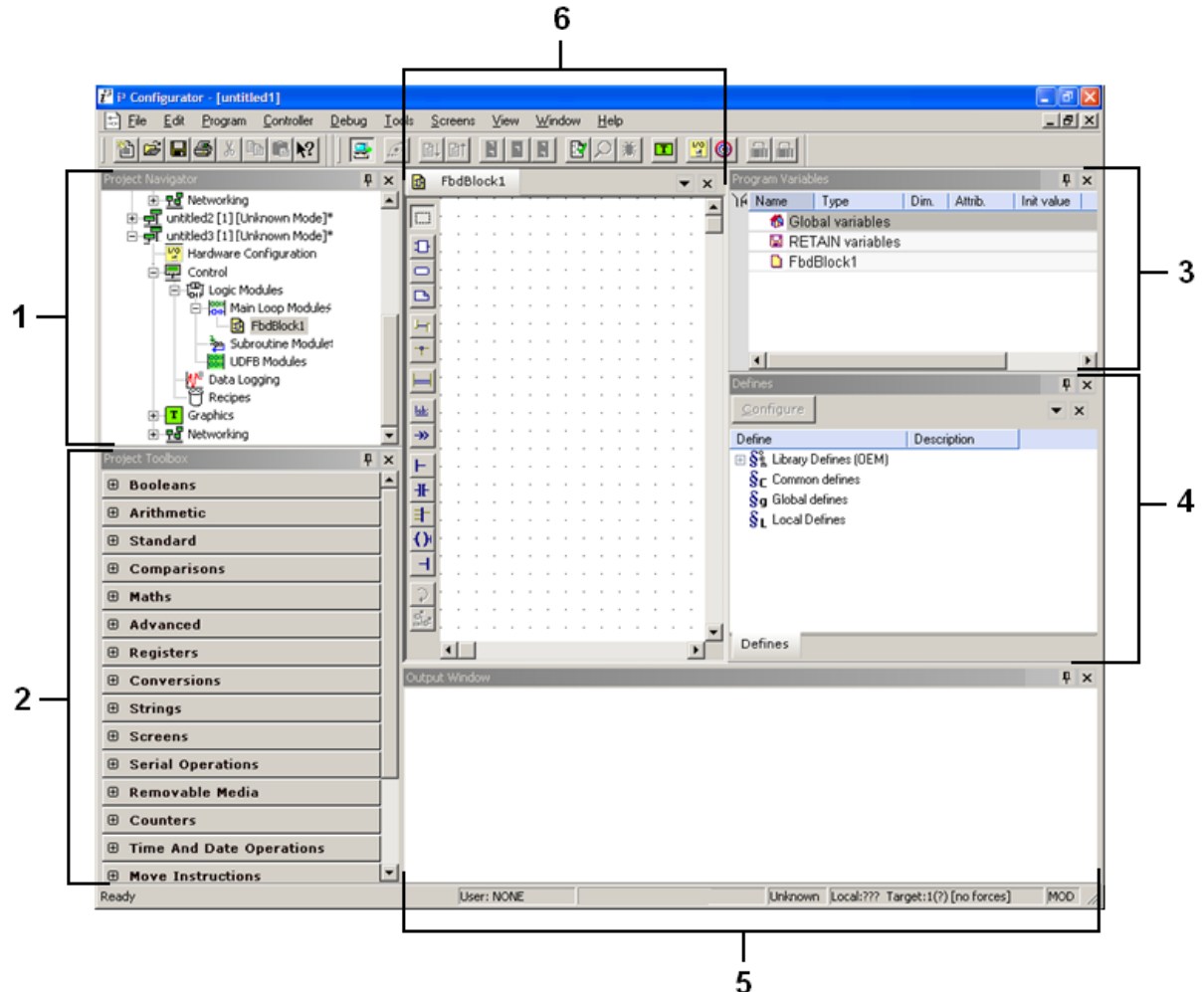
Integrated editing environment

Data types

Access to a bit in an integer

## The Main Window

The below dialog shows the workspace windows used when editing IEC programs. Individual component windows can be enabled/disabled by selecting them in the Tools Menu.



The following are the component windows:

1. Project Navigator: Similar in function to the Advanced Ladder Project Navigator
2. Project (IEC) Toolbox: Similar in function to the Advanced Ladder Project Toolbox
3. Program Variables: Lists out Variables used in the various IEC modules
4. Defines Window: A window in which constant values can be assigned or picked – includes system constants
5. Project Output Window: A window in which list of the errors in the last compilation are shown
6. Logic Editing Area

The following user interface items are used with the IEC editor:

**Tools Menu - Various component windows can be enabled/disabled from this menu.**

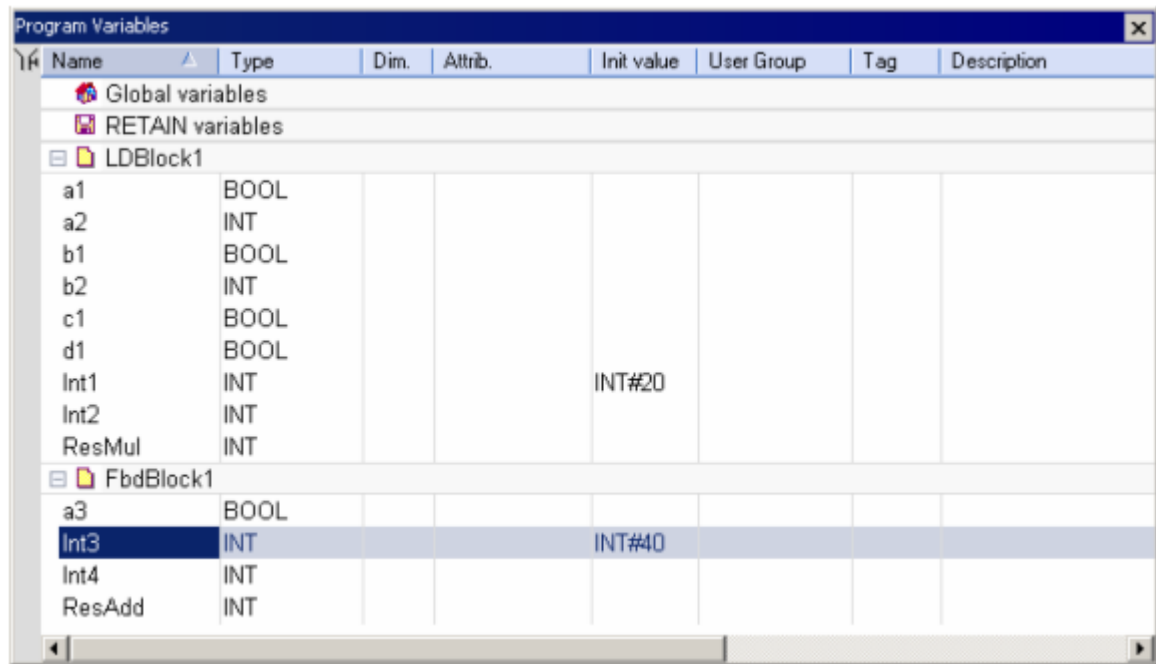
**IEC Editor logic modules toolbar can be used to create new logic blocks.**

## Declaring Variables


### *Program Variables Window*

The IEC Program Variables window contains a list of the variables used in the IEC Programming section.

**To open Program Variables window, enable Tools | Program Variables. This opens the following dockable Program Variables window.**



The columns of the Program Variables window can be made visible or invisible according to the requirement.

Double clicking on  icon to bring up the following dialog. Here the individual columns can be made visible or invisible by checking or unchecking the individual check box. The width of each column and its position can also be set here.



**Arrange columns** [X]

Column	Visible	Width	Filter
Name	<input checked="" type="checkbox"/>	100	
Type	<input checked="" type="checkbox"/>	80	
Dim.	<input checked="" type="checkbox"/>	40	
Attrib.	<input checked="" type="checkbox"/>	75	
Init value	<input checked="" type="checkbox"/>	60	
User Group	<input checked="" type="checkbox"/>	50	
Tag	<input checked="" type="checkbox"/>	50	
Description	<input checked="" type="checkbox"/>	500	
Value	<input type="checkbox"/>	70	

*Using the grid:*

Creating New Variables  
 Using the Editing Grid  
 Sorting Variables of a Group  
 Editing as Text  
 Bookmarks

*Each variable is described with:*

- Name
- Data Type and a Dimension
- Attribute
- Initial Value
- Tag and a Description Text

### ***Attributes of a Variable***

Each variable have an attribute displayed in the corresponding column of the grid. For each internal variable, you can select the "**Read Only**". Otherwise, the "attribute" column of an internal variable is empty. Parameters of UDFBs are marked as either "**IN**" or "**OUT**".

To change the attribute of an internal variable, enable the modification mode in the grid and move the cursor to the selected "attribute" cell. Then press ENTER to set or reset the "Read Only" attribute.

### ***Creating New Variables***

Hit INSERT key in the Program Variables Window to create a new variable in the selected group. The variable is added at the end of the group. Variables are created with a default name. You can rename a new variable or change its attribute using the variable editing grid.

In case of a group corresponding to local variables of a UDFB, pressing the INSERT key gives you the choice between:

- adding an "IN" (input) parameter
- adding an "OUT" (output) parameter
- adding a private variable

IN and OUT parameters always appear at the beginning of a UDFB group.

### ***Defining Structures***

To create a new type of data structure, right click in the program variables window and select Add Structure.

Each structure is represented as a group in the window. Enter the members of the structure in its group in the same way you enter variables in another group. New data structures are created with default name. Use right click option 'Rename' to change its name.

An instance of the structure can be created within another structure if the structure is already defined.

### ***Initial Value of a Variable***

A variable may have an initial value. The value must be a valid constant expression that fits to the data type of the variable. The initial value is displayed in red if it is not a valid expression for the selected data type.

There is no initial value for arrays and instances of function blocks.

To change the initial value of a variable, enable the modification mode in the grid and move the cursor to the selected "init value" cell. Then press ENTER to enter the new value.

### ***Naming a Variable***

To change the name of the variable, enable the modification mode in the grid and move the cursor to the selected "name" cell. Then press ENTER or hit the first character of the new name. Name is entered in a small box. Hit ENTER to validate the name or ESCAPE to cancel the change.

A variable must be identified by a unique name within its parent group. The variable name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A variable should not have the same name as a program or a user defined function block.

The name of a variable should begin by a letter or an underscore ("\_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a variable name. Naming is case insensitive. Two names with different cases are considered as the same.

### ***Sorting Variables***

At any moment you can sort variables of a group according to their name, type or dimension. For that you simply need to:

1. move the cursor to the header of the group
2. click on the name of the wished column

**i<sup>3</sup> Configurator** always keeps the original order of declared variables. Each time you insert a new variable or expand/collapse a group, the original sorting is re-applied.

### ***Variable Tag and Description***

**i<sup>3</sup> Configurator** enables you to freely enter for each variable two attributes that describe the variable:

- The "Tag" is the address of the variable that can be displayed together with the variable name in graphic languages. The variable address is associated with the Memory Area of the **i<sup>3</sup>**.
- The "Description" is a long comment text that describes the variable.

To change the tag or description of a variable, enable the modification mode in the grid and move the cursor to the corresponding cell. Then press ENTER to enter the new text.



### ***Variable Data Type and Dimension***

To change the type and dimension of the variable, enable the modification mode in the grid and move the cursor to the appropriate cell and press ENTER.

Each variable must have a valid data type. It can be either a basic data type or a type of function block or UDFB.

If the selected data type is STRING, you must specify a maximum length, and cannot exceed 255 characters.

Arrays: you can specify dimension(s) for an internal variable, in order to declare an array. Arrays have at most 3 dimensions. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by *ArrayName[0]*. You cannot declare arrays of function block instances. The total number of items in an array (merging all dimensions) cannot exceed 65535.

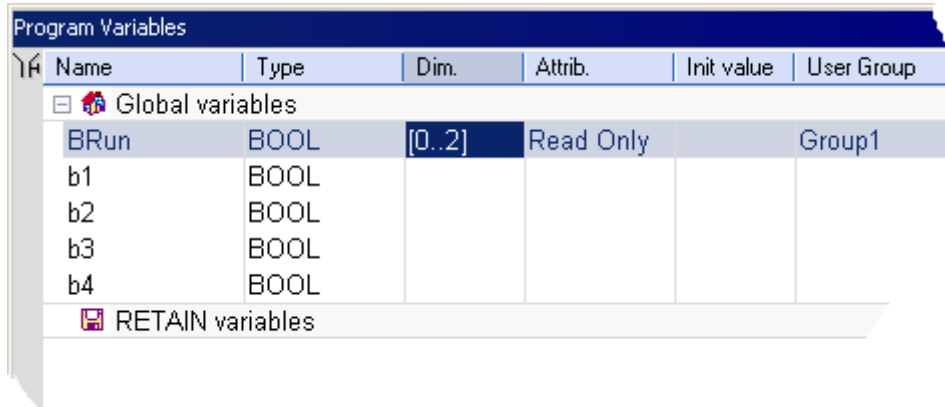
Using the Program Variables Window, you must enter integer dimensions separated by comas. For instance:

3,10,5

### Variable List - Active Grid

Hit SPACE bar to enable or disable the active grid.

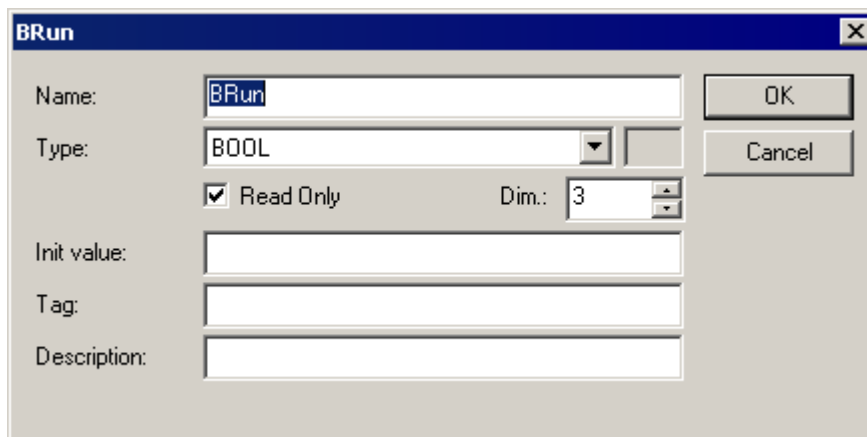
The Program Variables Window enables you to enter information in each cell of the active grid. At any moment, the active grid can be activated (each cell can be edited independently) or disabled (full row is highlighted) by selecting the variable and pressing space. It can also be done through the 'Enable Changes' right click option of the selected variable.



The screenshot shows a window titled "Program Variables" with a table of variables. The table has columns: Name, Type, Dim., Attrb., Init value, and User Group. There are two sections: "Global variables" and "RETAIN variables".

Name	Type	Dim.	Attrb.	Init value	User Group
<b>Global variables</b>					
BRun	BOOL	[0..2]	Read Only		Group1
b1	BOOL				
b2	BOOL				
b3	BOOL				
b4	BOOL				
<b>RETAIN variables</b>					

Press the ENTER key when the grid is inactive to open the variable setting box.



The screenshot shows a dialog box titled "BRun" with the following fields and controls:

- Name: BRun
- Type: BOOL (dropdown menu)
- ☒ Read Only
- Dim.: 3 (spin box)
- Init value: (text field)
- Tag: (text field)
- Description: (text field)
- Buttons: OK, Cancel

At any time you can drag with the mouse column separators in the main grid header for resizing columns.

Press the following keys for browsing groups of variables:

- |                  |   |
|------------------|---|
| Ctrl + Page Up   | Move the selection to the head of the previous group  |
| Ctrl + Page Down | Move the selection to the head of the following group |

## ***Editing Variables as Text***

### **Editing Variables as Text**


As an alternative to the user friendly grid for editing variables, it is possible to declare variables as text. Text editing applies to all the variables of a group. During text editing, the group and all its variables are locked in the grid so that no change can be entered from other windows.

Sereval syntaxes are available for describing variables:

IEC61131-3	The original IEC61131-3 syntax for declaring variables
XML tags	An easy XML structure using tags and attributes
CSV	CSV format (separator: semicolon)

To edit a group of variables as text, select the group in the program variables window. Right click and select 'Edit Variables as text'.

The Program Variables window goes blank and Logic editing area shows 3 tabs namely IEC format, XML format and CSV format. Select the tab in which you want to add variables.

On pressing '' in the logic editing area, will ask you if you wish to save. If the variables are saved, the same will be shown in Program Variables window.

To edit a group of variable as text, select the corresponding tab ("IEC", "XML" ou "CSV") at the bottom of the editing window, and then double click on the group name in the explorer pane.

### Editing Variables as Text Using IEC61131-3 Syntax

Using IEC61131-3 syntax, variables are declared within structured blocks. Each block begins with "VAR", "VAR\_INPUT", "VAR\_OUTPUT" or "VAR\_EXTERNAL" keyword and ending with "END\_VAR" keyword (with no semicolon after). Below is the meaning of each keyword:

<b>VAR</b>	Memory variables. Can be global, local or RETAIN depending on the edited group
<b>VAR_INPUT</b>	Input parameters of a block. Available only when the edited group is a UDFB.
<b>VAR_OUTPUT</b>	Output parameters of a block. Available only when the edited group is a UDFB.
<b>VAR_EXTERNAL</b>	External variables. Can be global or local depending on the edited group

#### Basic Syntax for Declaring a Variable:

To declare a variable, simply enter its symbol, followed by ":" and its data type. If the data type is STRING, it must be followed the maximum length between parenthesis.

Example:

```
MyVar : BOOL;  
MyString : STRING(255);
```

To indicate that a variable has the "read only" attribute, insert the "CONSTANT" keyword at the beginning of the variable declaration:

```
CONSTANT VarName : DataType;
```

To declare an array, the data type must be preceded by "ARRAY [ dimensions ] OF". There are at most 3 dimensions, separated by commas. Each dimension is specified as "0 .. MaxBound". Below are examples:

```
Array1 : ARRAY [0 .. 99] OF DINT;  
Matrix : ARRAY [0 .. 9, 0 .. 9, 0 .. 9] OF REAL;
```

Additionally, you can specify an initial value for single variables. The initial value is entered after the data type, and is preceded by ":= ". The initial value must be a valid constant expression that fits the data type.

Examples:

```
MyBool : BOOL := TRUE;  
MyString : STRING(80) := 'Hello';  
MyLongReal : LREAL := lreal#1.0E300;
```

Additional Information and Description Texts:

As a variable may have additional properties and comment texts in *i<sup>3</sup> Configurator*, we use special directives entered as IEC comments AFTER the declaration of the variable, to specify additional info. The following directives are available:

<b>(*\$tag=Text*)</b>	Variable tag (short comment)
<b>(*\$desc=Text*)</b>	Variable description

You can also use "/" single line comments to enter the directives:

```
// $tag=Text  
// $desc=Text
```

### Editing Variables as XML Tags

You can describe variable using a simple XML structure, where each variable is described as an XML tag. The file must fit the basic XML syntax. Values of tag attributes must be entered between double quotes. Characters < > " ' & are reserved to XML and cannot appear in values of tag attributes. Instead you should use the following sequences:

<	&lt;
>	&gt;
"	&quot;
'	&apos;
&	&amp;

Below is the tag structure for variable declaration:

```
<k5project>
|
+-<vargroup>
|
+-<var>*
|
+-<varinfo>*
```

(the "\*" mark indicates that the tag can appear 0 or more times)

The rest of this page describes the format and meaning of each tag:

#### **<k5project>**

This tag must be entered at the top level and is unique. It is reserved for extensions (enhancement of the XML structure), and specifies the version of the syntax. Its attributes are:

<b>version</b>	Reserved for future extensions. This attribute is mandatory and must be "1.0".
----------------	---

The <K5Project> tag contains one <vargroup> tag.

#### **<vargroup>**

This tag must appear with the <K5Project>, and contains all <var> tags for variables of the group. In this version, the tag has no attribute (the name of the group is implicit)

#### **<var>**

This tag describes the basic definition of one variable. Its attributes are:

<b>name</b>	Symbol of the variable. This attribute is mandatory.
<b>type</b>	Name of the data type of the variable This attribute is mandatory
<b>len</b>	Maximum length if the data type is STRING. This attribute is mandatory for STRING variables, and should not appear for other data types.
<b>dim</b>	Dimension(s) if the variable is an array. There are at most 3 dimensions, seperated by comas. This attribute is optional.
<b>attr</b>	Attributes of the variable, separated by comas. Possible values are: <b>IN</b> : this is an INPUT parameter (for UDFBs only) <b>OUT</b> : this is an OUTPUT parameter (for UDFBs only) <b>external</b> : this is an external variable <b>constant</b> : variable is read only This attribute is optional.
<b>init</b>	Initial value of the variable Must be a valid constant expression that fits the data type This attribute is optional

The <var> tag contains zero or more <varinfo> tags.

<b>&lt;varinfo&gt;</b>
------------------------

This tag indicates an additional info for the variable it belongs to. Its attributes are:

<b>type</b>	Type of information contained in the " <b>data</b> " attribute. Possible values are: <b>tag</b> : variable tag (short comment) <b>desc</b> : description This attribute is mandatory.
<b>data</b>	Data specified by the " <b>type</b> " attribute, in text format. This attribute is mandatory

### Editing variables as text in CSV format

Using CSV format, each variable is defined on one line of text. Each component of the variable definition is entered as one CSV element. CSV elements are separated by semi-colons. Each element is written between double quotes. A double quote within an element is represented by two double quotes. CSV format is an easy way to exchange variable declaration with Spreadsheet applications.

It is not mandatory that all elements (all columns) appear in the text. The first line must contain the list of columns used, using the following keywords:

<b>name</b>	variable symbol this item is mandatory
<b>type</b>	name of the data type this item is mandatory, and must appear before len, dim and init columns
<b>len</b>	string length if the data type is STRING this item must be empty for other data types
<b>dim</b>	dimensions in case of an array there are at most 3 dimensions, separated by comas
<b>attr</b>	attribute of the variable, can be: <b>IN</b> : input parameter of a UDFB <b>OUT</b> : output parameter of a sub-program <b>external</b> : extern variable
<b>RO</b>	if "YES" indicates that the variable has the read-only attribute (note: you can also use "TRUE" or "1" value)
<b>init</b>	initial value of the variable must be a valid constant expression that fits the data type
<b>tag</b>	tag (short description text)
<b>desc</b>	description text

Below is an example of CSV text for the declaration of 3 variables, with some columns missing:

```
"name","type","len","attr","RO"  
"MyVar","BOOL","","","NO"  
"ExtVar","DINT","","external","YES"  
"MyStr","STRING","10","","NO"
```



## Editing Programs

### *Editing Programs*

The Programming environment provide language dedicated editors for:

Sequential Function Chart (SFC)

Function Block Diagram (FBD)

Ladder Diagram (LD)

Structure Text (ST)

Instruction List (IL)

Please refer to the following topics for common features:

Selecting function blocks

Selecting variables and instances

Quick Search

Bookmarks

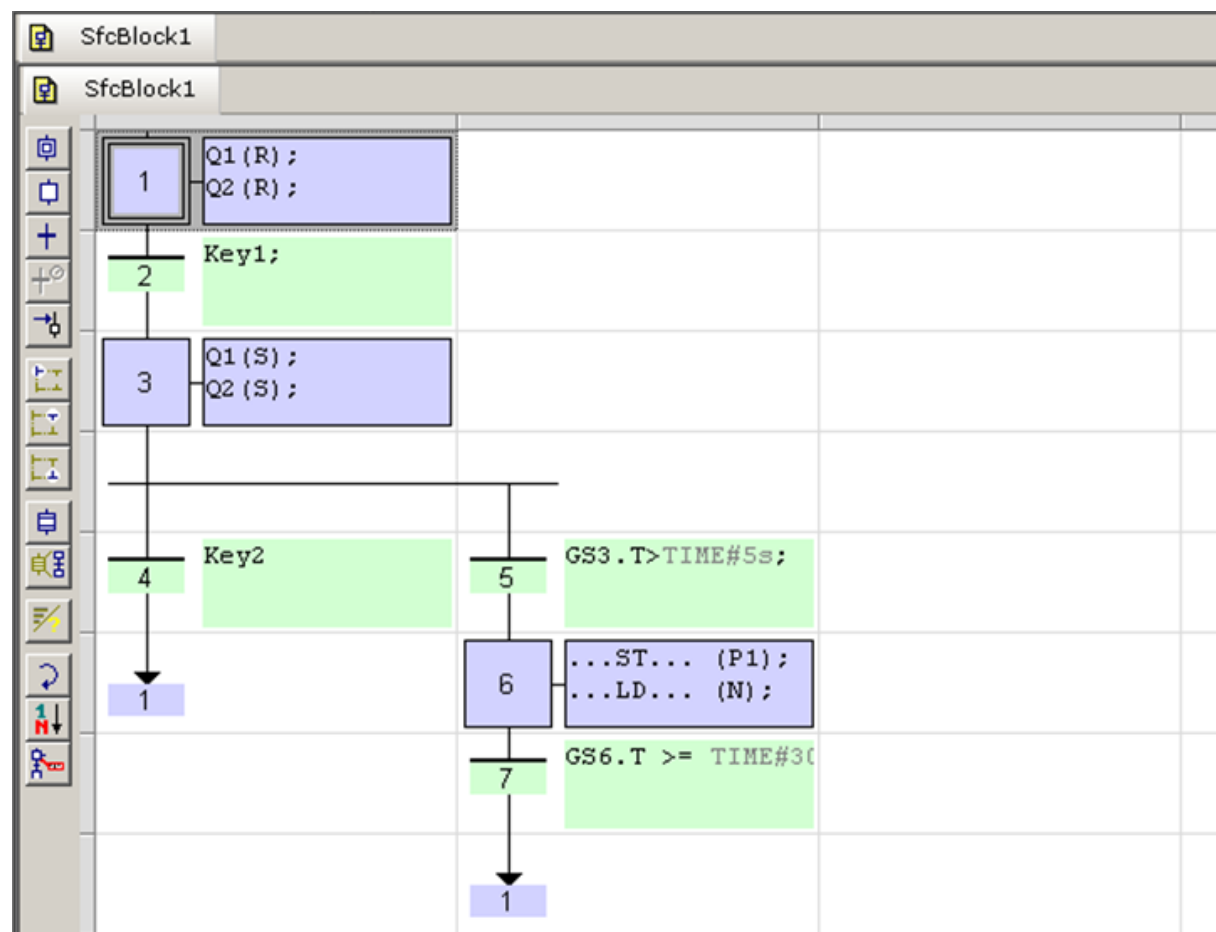
The editor provides you the ideal programming environment with drag and drop features:

- drag a variable from the list to the program to insert it
- drag a definition to the program to insert its name
- drag a block in the program to insert it
- drag a function block to the variable list to declare an instance
- drag a variable from the program or from the variable list to the spy list
- double click on a line of the output window to highlight the corresponding code...

## SFC Editor

### Sequential Function Charts (SFC) Editor

The SFC editor is a powerful graphical tool that enables you to enter and manage Sequential Function Charts according to the IEC 61131-3 standard. The editor supports advanced graphic features such as drag and drop, so that you can rapidly and freely arrange the elements of your diagram. It also supports automatic chart formatting when inserting or deleting items and thus enables quick input using the keyboard.



#### SFC Diagram components:

- Steps
- Transitions
- Divergences
- Parallel branches
- Jump to a step
- Macro steps
- Actions
- Conditions

#### Related Sections:















- Using the SFC toolbar
- Drawing divergences
- Viewing the chart
- Moving or copying parts of the chart
- Entering macro-steps
- Renumbering steps and transitions
- Entering actions of a step
- Entering condition of a transition
- Notes for steps and transitions
- Bookmarks

Tips:

- To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.
- Hit SPACE bar on the main corner (on the left) of a divergence or convergence to set double/single horizontal line style.

## Using the SFC Toolbar


The vertical toolbar on the left side of the editor contains buttons for inserting items in the chart. Items are always inserted before the selected item. The chart is automatically re-arranged when a new item is inserted.

	<b>Initial Step[I]</b> : Insert an initial step
	<b>Step[S]</b> : Insert a Step
	<b>Transition[T]</b> : Insert a transition
	<b>Set timer on transition</b> : Set timer to a transition
	<b>Jump[J]</b> : Insert a jump to a step
	<b>Divergence[X]</b> : Insert the main (left side) corner of a divergence or convergence
	<b>Divergence[D]</b> : Insert a divergence corner
	<b>Convergence[C]</b> : Insert a convergence corner
	<b>Macro[M]</b> : Insert a macro-step
	<b>Macro Body[B]</b> : Insert the body of a macro-step
	<b>Swap item style(Space):</b> Swap between possible overviews: display code of actions and conditions  display notes attached to steps and transitions
	<b>Step Selection</b> : Switch between Initial Step or a Step. This is available for Steps only
	<b>Renumber</b> : Assigns number in order.
	<b>Edit reference</b> : New reference can be assigned.

Use the following keyboard commands when an item is selected:

- **ENTER**: edit the level 2 of a step or transition
- **Ctrl+ENTER**: change the number of a step, transition or jump




The last button of the toolbar enables you to switch between possible displays:

- Swap between possible overviews of level 2 in the level 1 chart:
- |   |  |
|---|--|
|  | <ul style="list-style-type: none"><li>• display code of actions and conditions</li><li>• display notes attached to steps and transitions</li></ul> |
|---|--|

## Drawing SFC Divergences

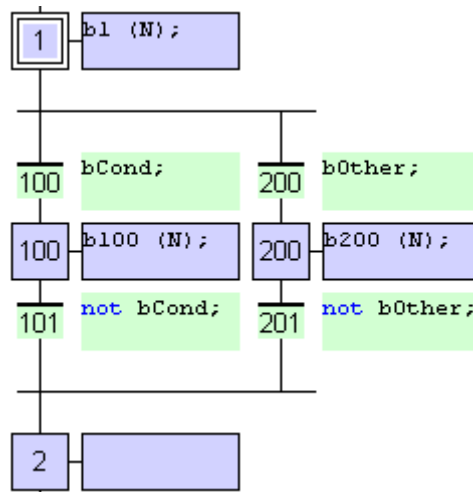
When using the SFC editor, you just need to place items in the grid. The editor calculates and draws lines automatically to link steps, transitions and jumps you place in the chart.

The same method is used for drawing divergences: you just need to place the "corners" that identifies divergences, convergences and branches. The editor takes care of drawing vertical and horizontal lines. Use the following buttons in the SFC toolbar:

-  Insert the main (left side) corner of a divergence or convergence
-  Insert a divergence corner
-  Insert a convergence corner

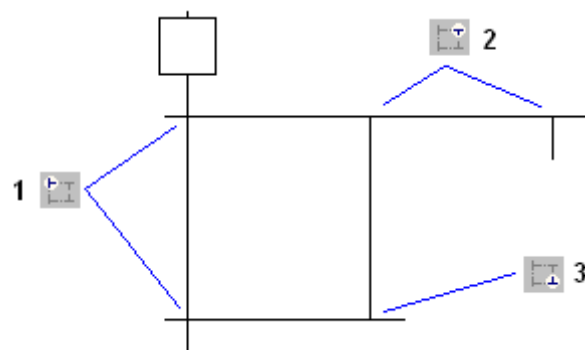
Important Note:

Divergences are always drawn from the left to the right. The first branch, on the left, contains the "corners" that identify the divergence. It must be aligned with the preceeding step or transition:



How to Proceed:

1. Insert the main corner (on the left side branch) of the divergence and the convergence
2. Insert corners at the top of each branch (divergence)
3. Insert corners at the bottom of the branches where a divergence is wished



Simple or Double Divergence Lines:

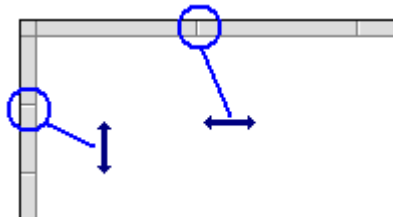
You can change the drawing of a divergence or convergence horizontal line, for drawing simple or double lines according to SFC definition. To do that, move the selection on the main corner (on the left) and hit the SPACE bar.

### Viewing SFC Charts


The chart is entered in a logical grid. All objects are snapped to the grid.

At any moment you can use the commands of the "View" menu for zooming in or out the edited chart. You also can press the [+] and [-] keys of the numerical keypad for zooming the diagram in or out.

You also can drag the separation lines in vertical and horizontal rulers to freely resize the cells of the grid:



The SFC editor adjusts the size of the font according to the zoom ratio. When cells are wide enough, a text overview with the contents of the step or transition (level 2). The last button of the toolbar enables you to switch between possible displays:

-  Swap between possible overviews of level 2 in the level 1 chart:
- display code of actions and conditions
  - display notes attached to steps and transitions

### Moving and Copying SFC Charts

The SFC editor fully supports drag and drop for moving or copying items. To move items, select them and simply drag them to the desired position.

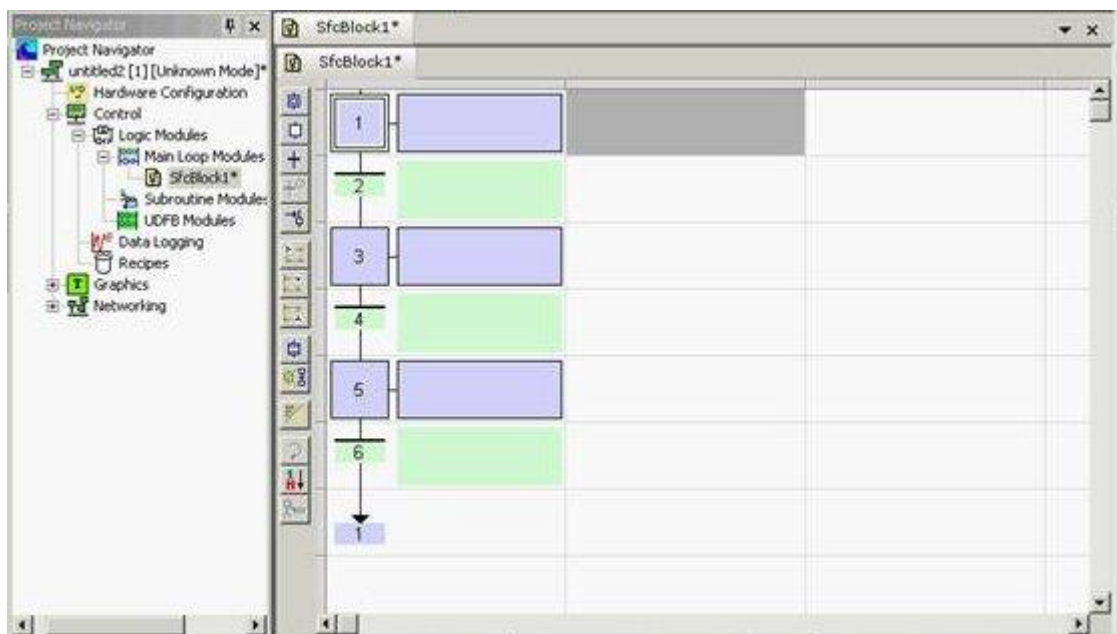
To copy items, you may do the same, and just press the CONTROL key while dragging. It is also possible to drag pieces of charts from a program to another if both are open and visible on the screen.

At any moment while dragging items you can press ESCAPE to cancel the operation.

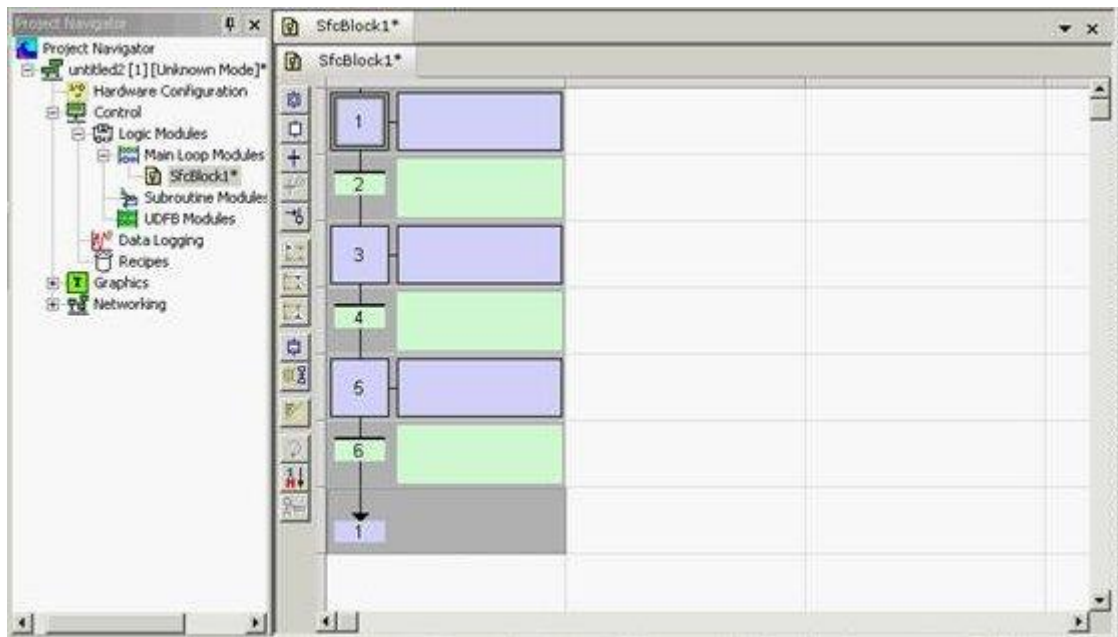
Alternatively, you can use classical Copy / Cut / Paste commands from the Edit menu. Paste is performed at the current position.

Steps to be followed to perform Drag/Drop Operation in IEC Program's SFC Editor:

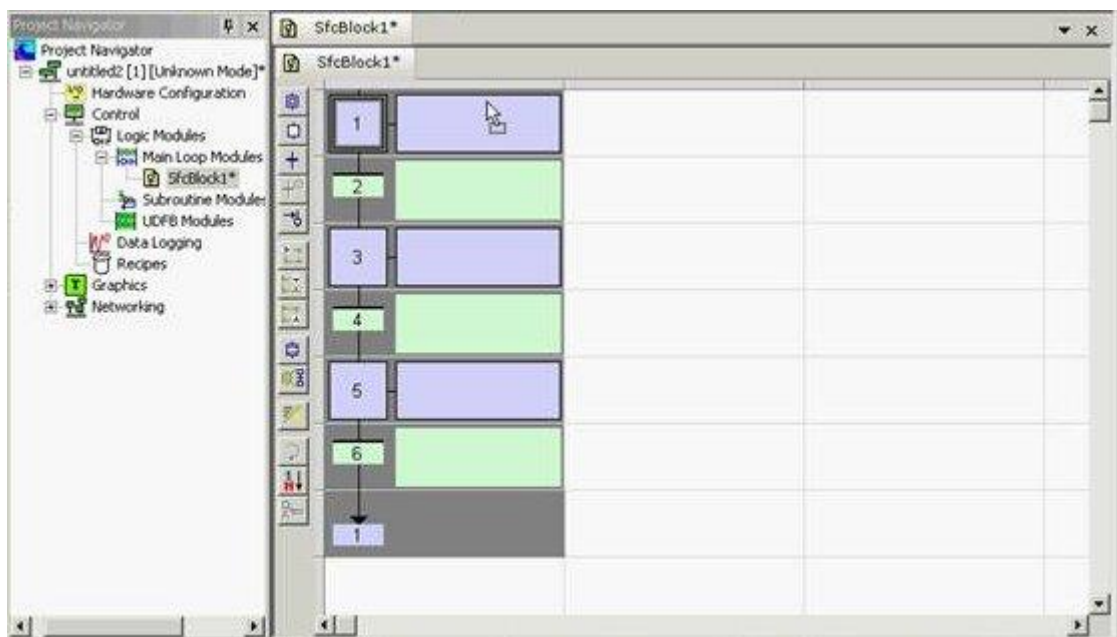
Step 1: Open the SFC Logic Module Editor whose logic has to be moved/copied.



Step 2: Select the logic section to be moved/copied.



Step 3: Bring the mouse cursor on the selection area and hold left mouse button in pressed state for couple of seconds until the selected area's background color changes and the cursor's shape changes.

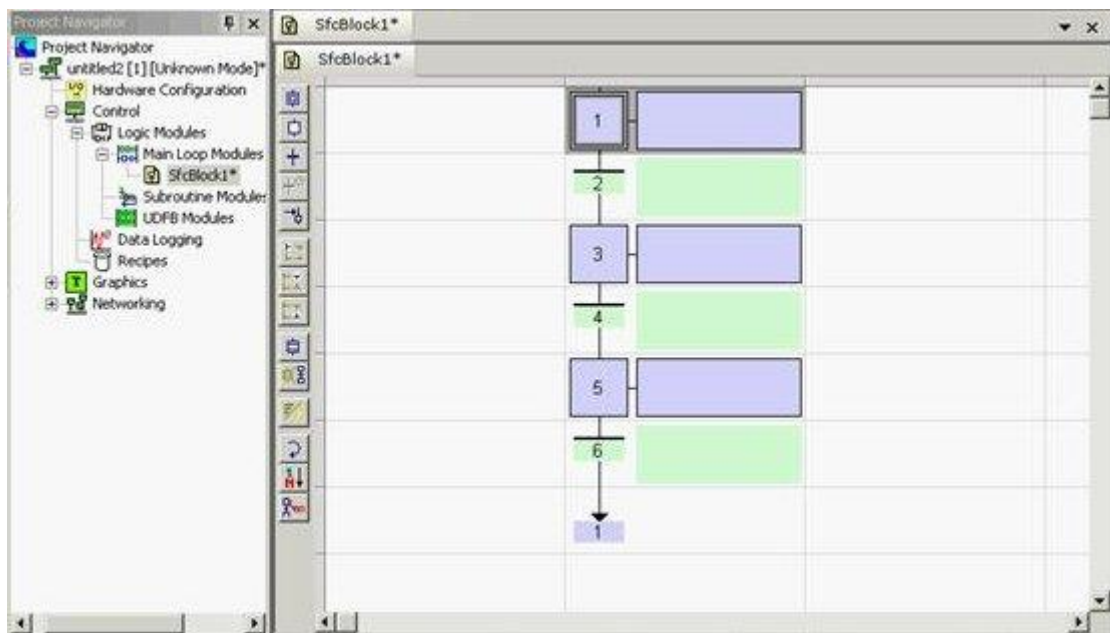


Step 4: With the mouse's left mouse button in pressed state move the mouse to move the selection to desired area.

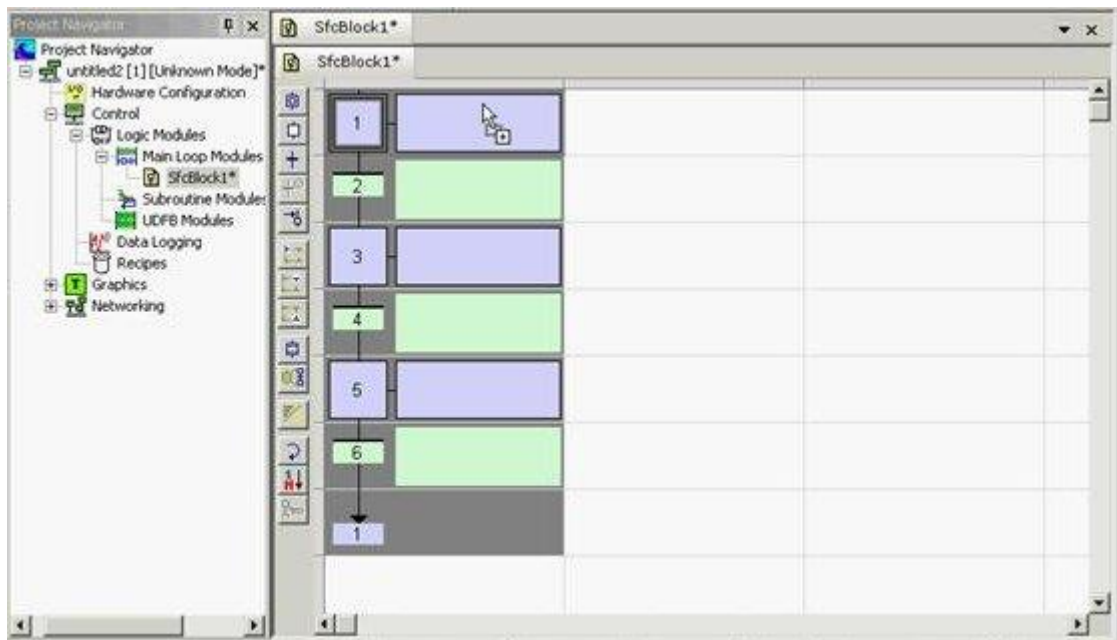




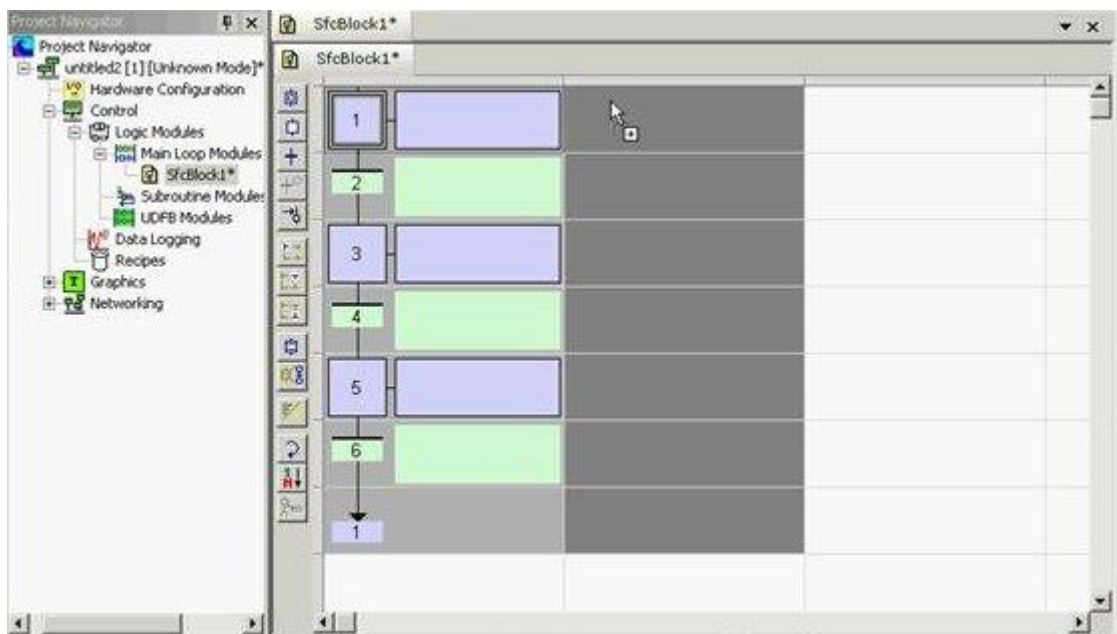
After releasing the mouse, the copy of the selected area must be placed in the desired location:



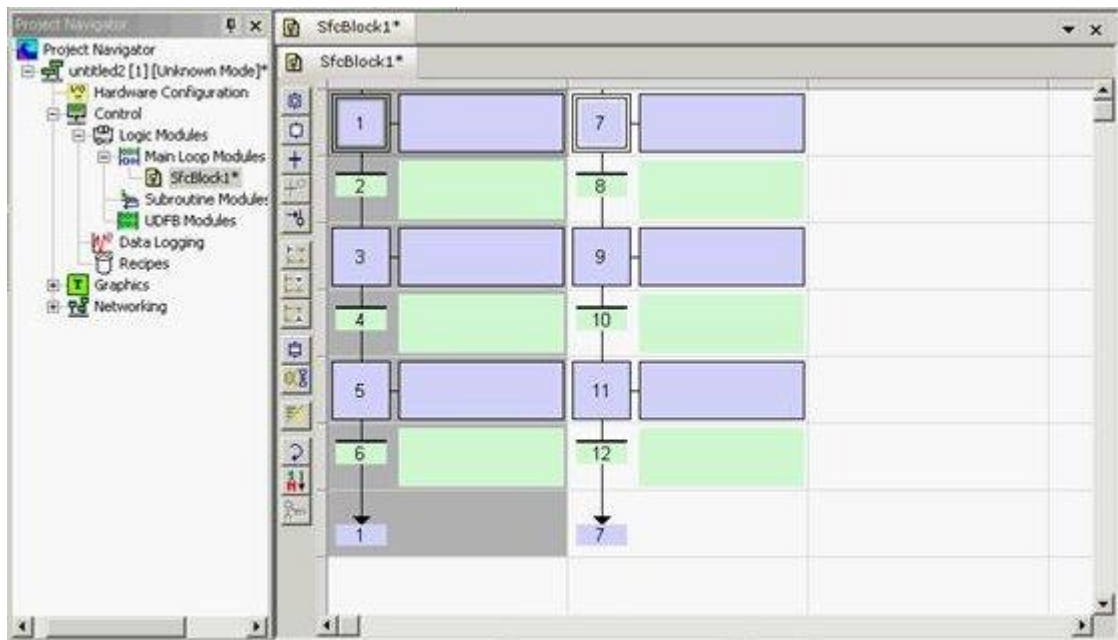
If the selection is to be copied, the hold the CONTROL Key in pressed state while moving and releasing the left mouse button.



With the mouse's left mouse button in pressed state move the mouse to move the selection to desired area.



After releasing the mouse, the copy of the selected area must be placed in the desired location:

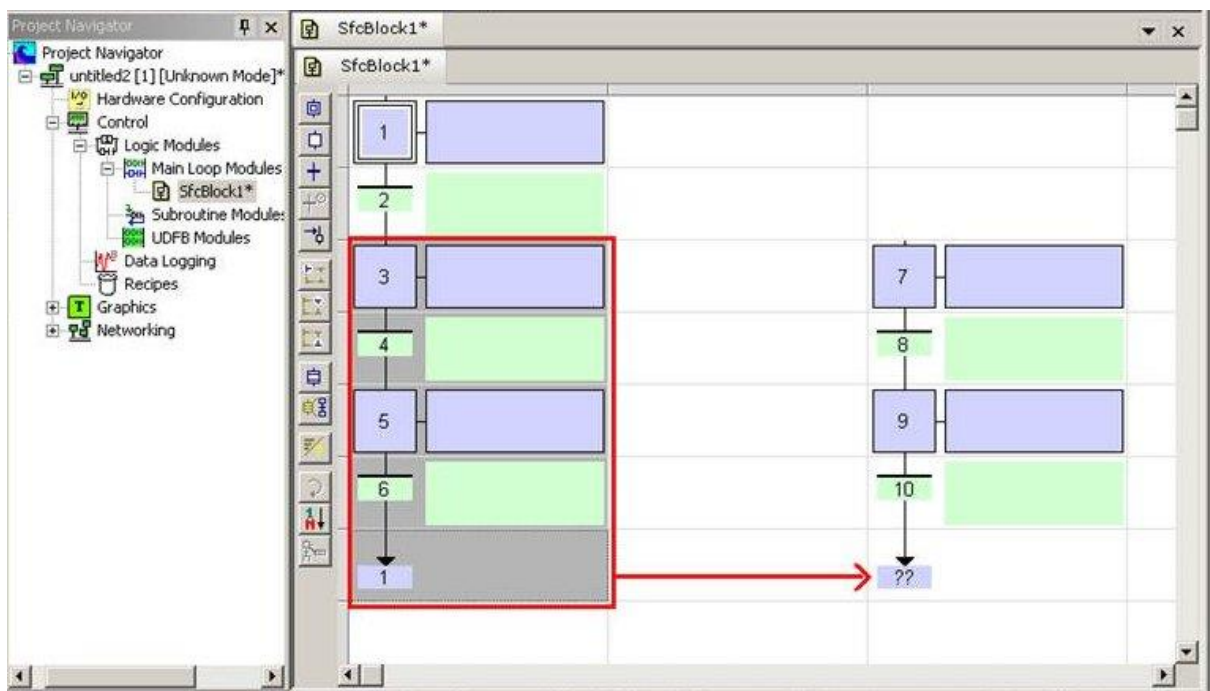


Steps to be followed to while copying SFC Module's Logic section that has Jump Block included in the selection:

Improper Selection

Selected area does not include Step#1 which the Jump block is referring to.

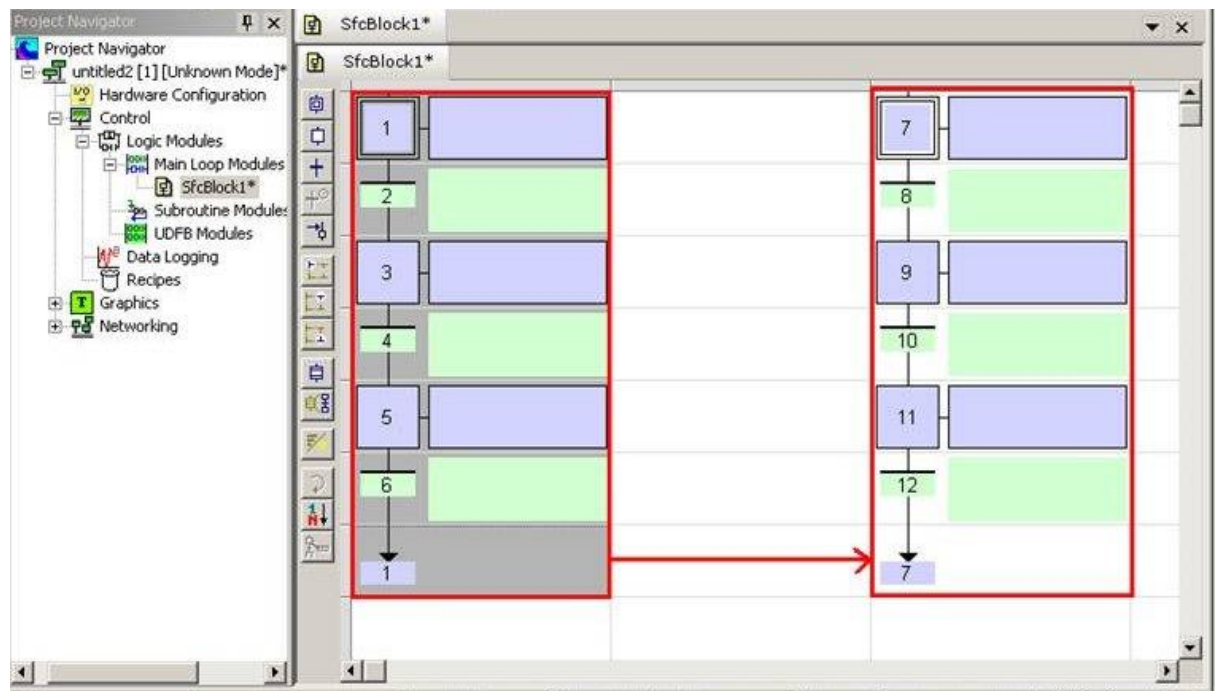
As a result, after Copy/Paste, Jump block's Jump value does not get updated.



Proper Selection:

Selected area includes the block to which the Jump block is referring to.

As a result, when a Copy/Paste operation is done, the Jump value is also updated



### Entering SFC macro-steps

A macro step is a special symbol that represents, within a SFC chart, a part of the chart that begins with a step and ends with a step. The body of the macro-step must be declared in the same program. The body of a macro-step begins with a special "begin" step with no link before, and ends with a special "end" step with no link after. The symbol of the macros step in the main chart has double horizontal lines.

Use the following buttons of the SFC toolbar for entering macro-steps:



Insert a macro-step



Insert the body of a macro-step

Important note: The symbol of the macro-step and the beginning step of its body must have the same number.

Hit Ctrl+ENTER when a macro-step symbol or a beginning step is selected to change its number.

### **Renumbering Steps and Transitions**

Each step or transition is identified by a number. A jump to a step is also identified by the number of the destination step. The SFC editor allocates a new number to each step or transition inserted in the chart.

To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.

It is not possible to change the number of a step or a transition if its level 2 is currently open for editing. The number is used for identifying the step or transition in the level 2 editing window.

In compiler reports, a step is identified by its number prefixed by "GS". A transition is identified by its number prefixed by "GT".

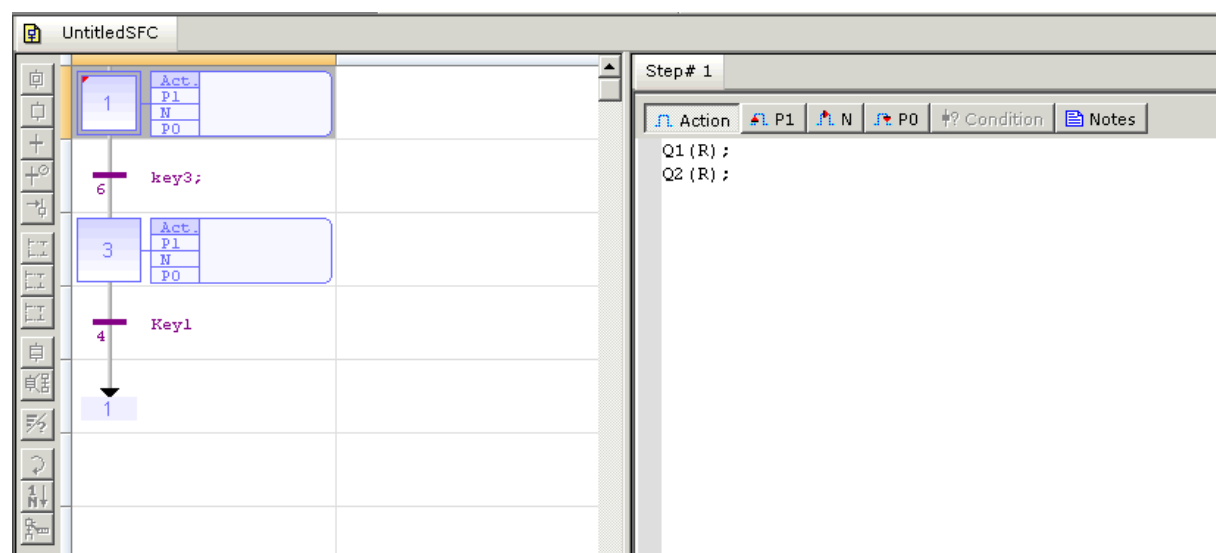
### Entering Actions of a Step

Actions and notes attached to a step (level 2) are entered in a separate window. To open the level 2 editing window of a step or transition, double click on its symbol in the chart, or select it and hit ENTER.

The level 2 editing window proposes 4 views for entering different types of level 2 information:

- simple actions entered as text
- P1 actions that can be programmed in ST/IL text, LD or FBD
- N actions that can be programmed in ST/IL text, LD or FBD
- P0 actions that can be programmed in ST/IL text, LD or FBD
- text notes

Use the tab buttons in the level 2 editing window for selecting a view:



The condition for Steps and Transitions of IEC SFC Editor can be made in the embedded conditions editor tab.

The editor for preferred condition of a step can be accessed by clicking the appropriate toolbar button.

The condition Action – as described in Action tab, P1 – Rising Edge, N – Every cycle, P0 – Falling Edge will be available for Steps. These conditions will be executed when the Step is activated. The descriptions are displayed in the box next to each step.

Action Condition – Only ST Language is Allowed.  
P1 Condition: ST/FBD/LD Languages are allowed  
N Condition: ST/FBD/LD Languages are allowed  
P0 Condition: ST/FBD/LD Languages are allowed.

The default setting of language used to edit conditions of a step or transition is set to ST Language. In order to change the type of language for a step, you can select it from the Edit menu.

**Note:** The language can be changed for a condition only if the condition contents are empty.

When editing P1, N or P0 actions, use the "Edit / Set Language" menu command to select the programming language. This command is not available if the action block is not empty.

The first view ("Action") contains all simple actions to control a boolean variable or a child SFC chart. However, it is possible to directly enter action blocks programmed in ST together with other actions in this view. Use the following syntax for entering ST action blocks in the first pane:

```
ACTION ( qualifier ) :  
    statements...  
END_ACTION;
```

Where *qualifier* is "P1", "N" or "P0".



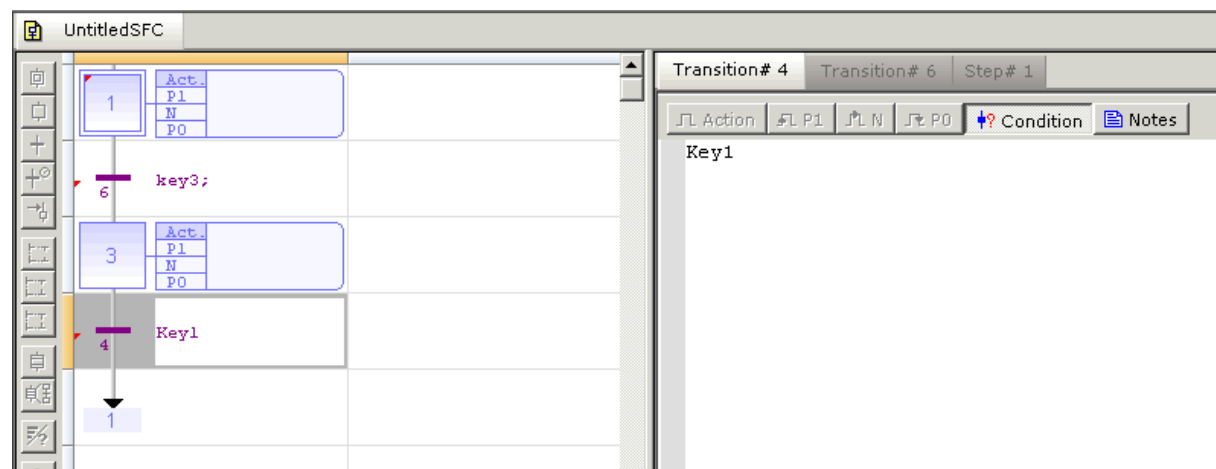
### Entering the Condition of a Transition

The condition and notes attached to a transition (level 2) are entered in a separate window. To open the level 2 editing window of a step or transition, double click on its symbol in the chart, or select it and hit ENTER.

The level 2 editing window proposes 2 views for entering different types of level 2 information:

- condition programmed in ST/IL text or LD
- text notes

Use the tab buttons in the level 2 editing window for selecting a view:



The languages that can be used to edit the conditions for a step or transition can be modified if that condition supports multiple languages.

The description Condition will be available only for Transitions. The descriptions are displayed in the box next to each step and transition.

Transition Condition: ST/LD Languages are allowed

**Note:** Only ST Language is allowed

The default setting of language used to edit conditions of a step or transition is set to ST Language. In order to change the type of language for a step, you can select it from the Edit menu.

**Note:** The language can be changed for a condition only if the condition contents are empty.

When editing the condition, use the "Edit / Set Language" menu command to select the programming language.

This command is not available if the condition is not empty. FBD cannot be used to program a condition.

### Entering Notes for Steps and Transitions

The SFC editor supports the definition of text notes for each step and transition. The notes are entered in the level2 editing window of steps and transitions. Refer to the following topics for further information about the level 2 editing window:

- Entering Level 2 for steps
- Entering Level 2 for transitions

Notes can be displayed in the chart. The last button of the toolbar enables you to switch between possible displays:

Swap between possible overviews of level 2 in the level 1 chart:



- display code of actions and conditions
- display notes attached to steps and transitions

Notes have no meaning for the execution of the chart. Entering notes for steps and transitions enables you to enhance the auto-documentation of your programs. It also provides an easy way to write and exchange specifications of an SFC program before actions and conditions are programmed.

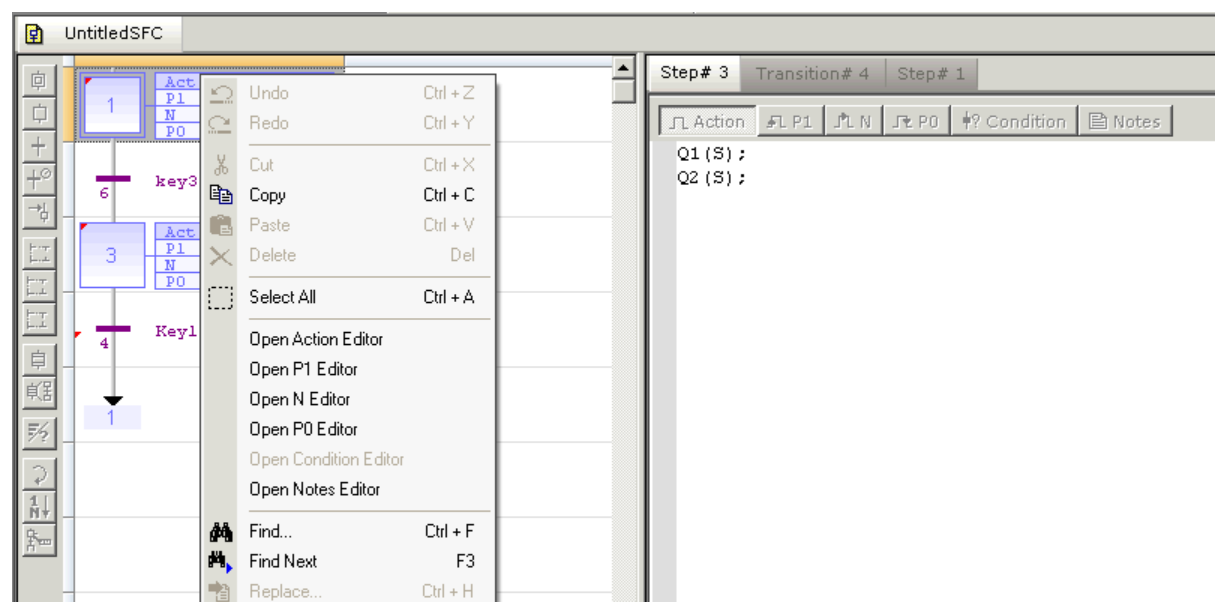
### Viewing SFC Logic and Secondary Editor Simultaneously

**i3 Configurator** version 9.10B supports accessing of main SFC Logic and its secondary editors simultaneously to the user.

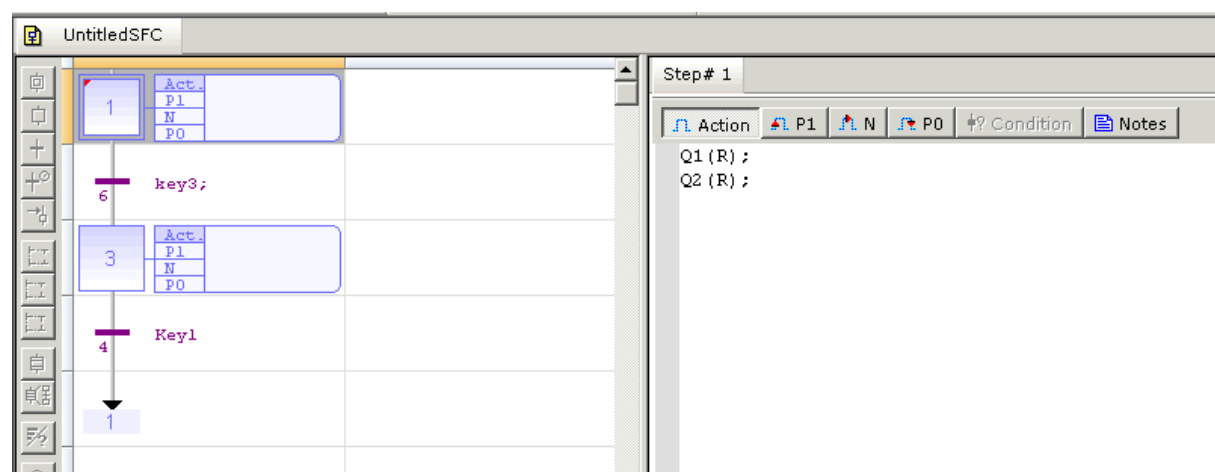
Access to SFC Logic and its secondary editors simultaneously:

The user can now view and edit the logic associated with multiple step/transition blocks of an SFC Logic Module simultaneously. The secondary editor can be opened in either of the following ways:

- Double clicking on a step/transition or select it and hit ENTER to open its associated editor
- Using right click options to open secondary editor



Once selected for viewing/editing, the step or transition will be opened in an editor window indicating the respective step/ transition name as a tab.



Opening multiple step/transition editors:

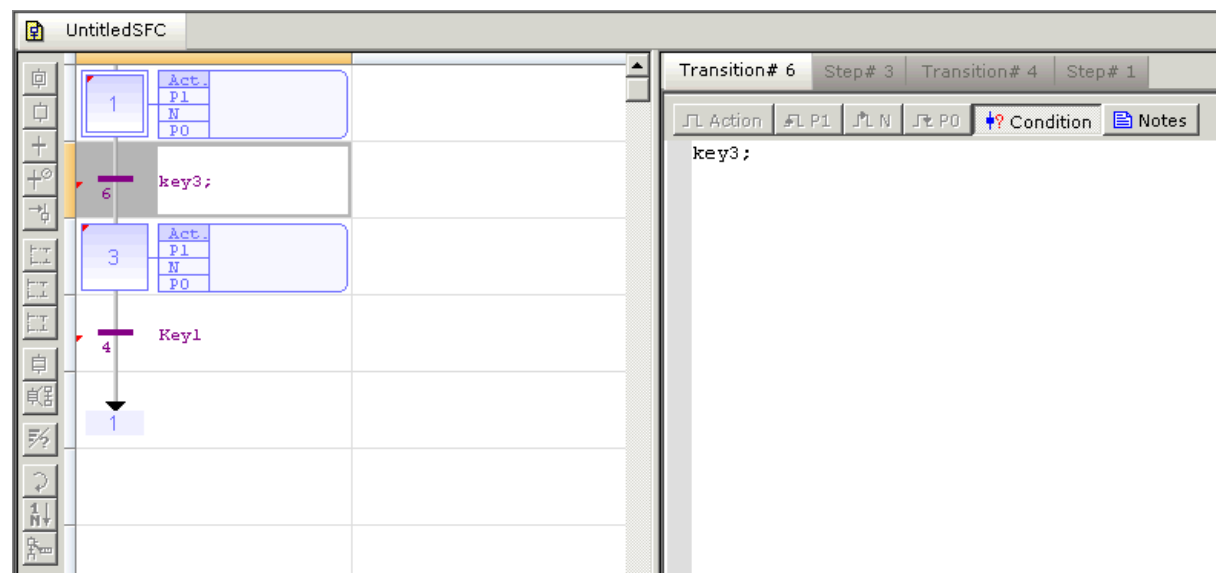
Multiple steps/transitions can be opened for viewing & editing in non-debug mode. Once opened the open steps/transitions are indicated with a lock status (red indication on the left hand corner of the step/transition).

When secondary editors set for one or more Step/Transition block has been opened, the main SFC Logic Editor will set to read-only mode. In this mode, the normal operations will be disabled along with the sticky toolbar. In this mode, user can only:

- a. Double-click on step/transition block or select it and hit ENTER to access its secondary editors' view set.
- b. Right-click on step/transition block to access the context menu.

Once all secondary editors' sets have been closed, the SFC Logic block will be set to normal mode.

Note: Closing of the respective step/transition removes the lock indication.



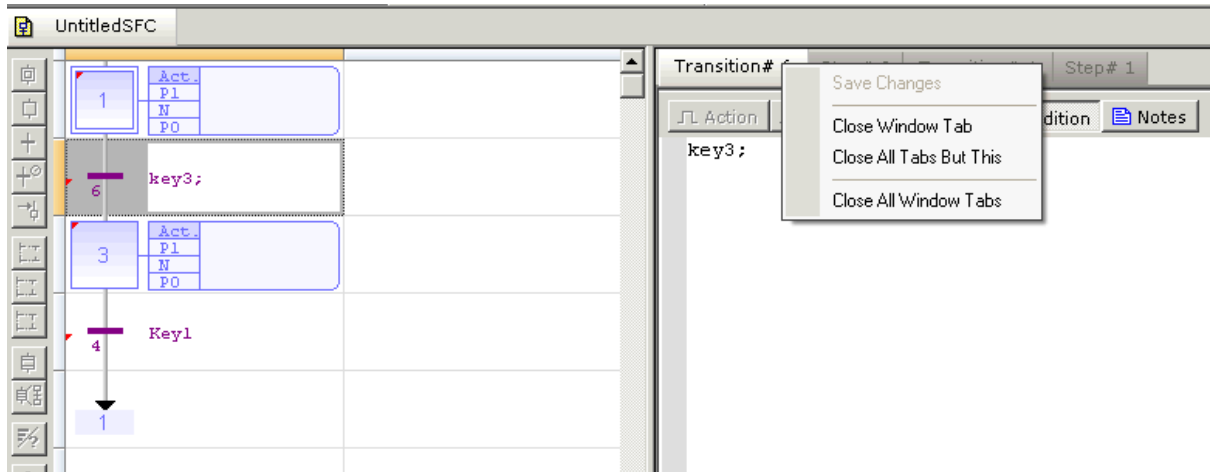
Right click options on tab name of secondary editors

The following right click options on tab name of secondary editors to save & close of Steps/Transitions are available to user.

Right click option on secondary editor tab include:

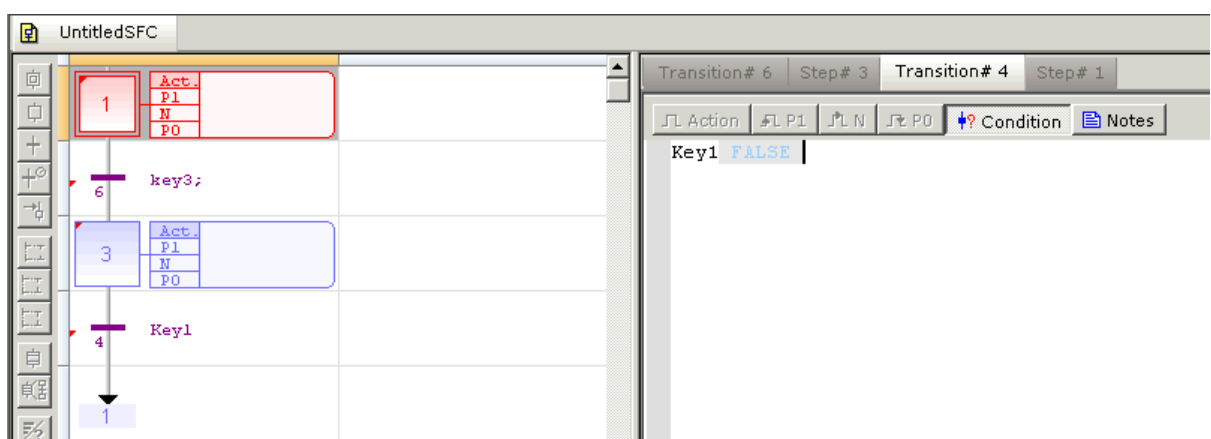
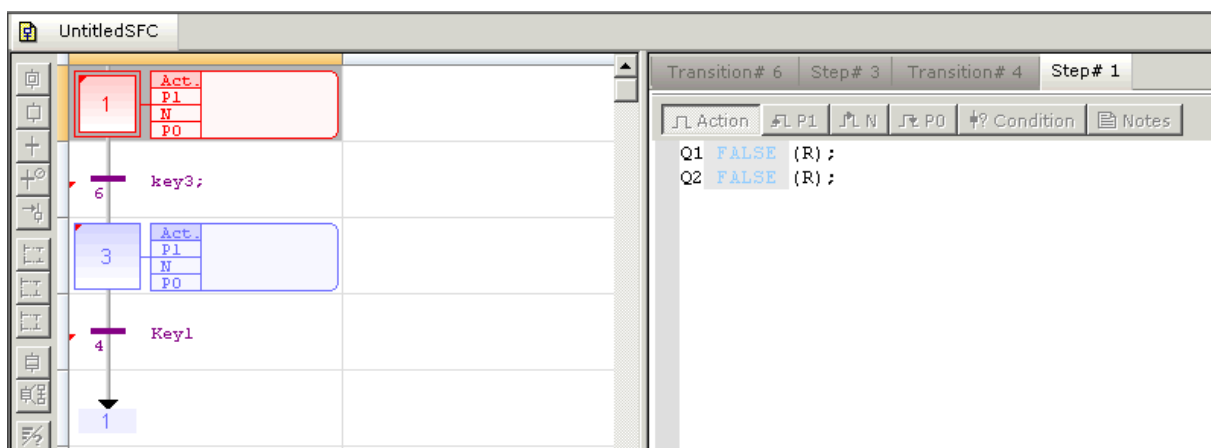
- Save Changes - Saves any changes.
- Close window tab - closes the selected tab
- Close all but this - closes all the opened tabs but the selected one
- Close all Window tab - closes all the opened tab.

Note: Save option is available only if changes have been made in the selected Step/Transition.



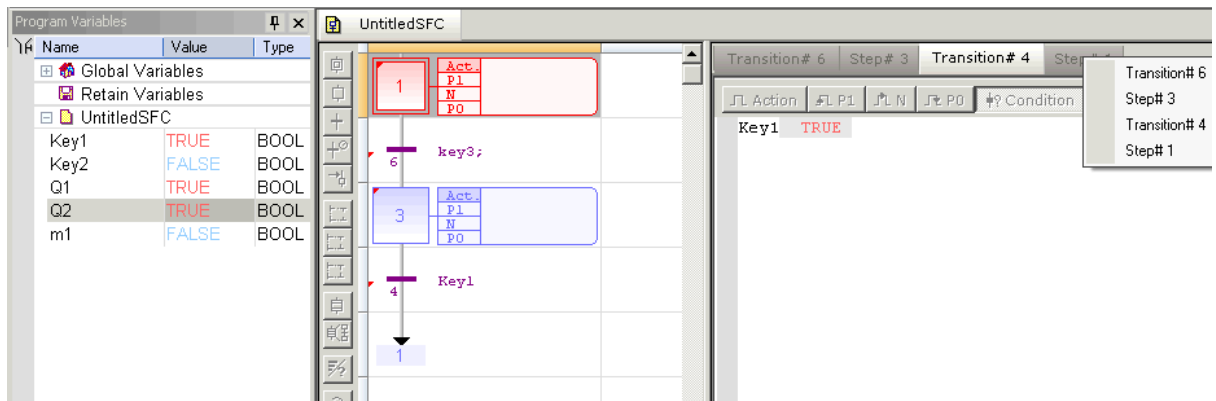
### Debugging Step/Transition logic

The user can debug the main SFC Logic as well as the secondary logic and view them simultaneously. The active step/transition in run is indicated with red color. In debug mode SFC logics & secondary editor are not allowed for editing. Multiple steps/transitions can be opened for viewing during debug mode.



### Selection of opened steps/ transitions

Multiple steps/ transitions opened can be selected from the drop down list and navigated in both debug & non-debug mode



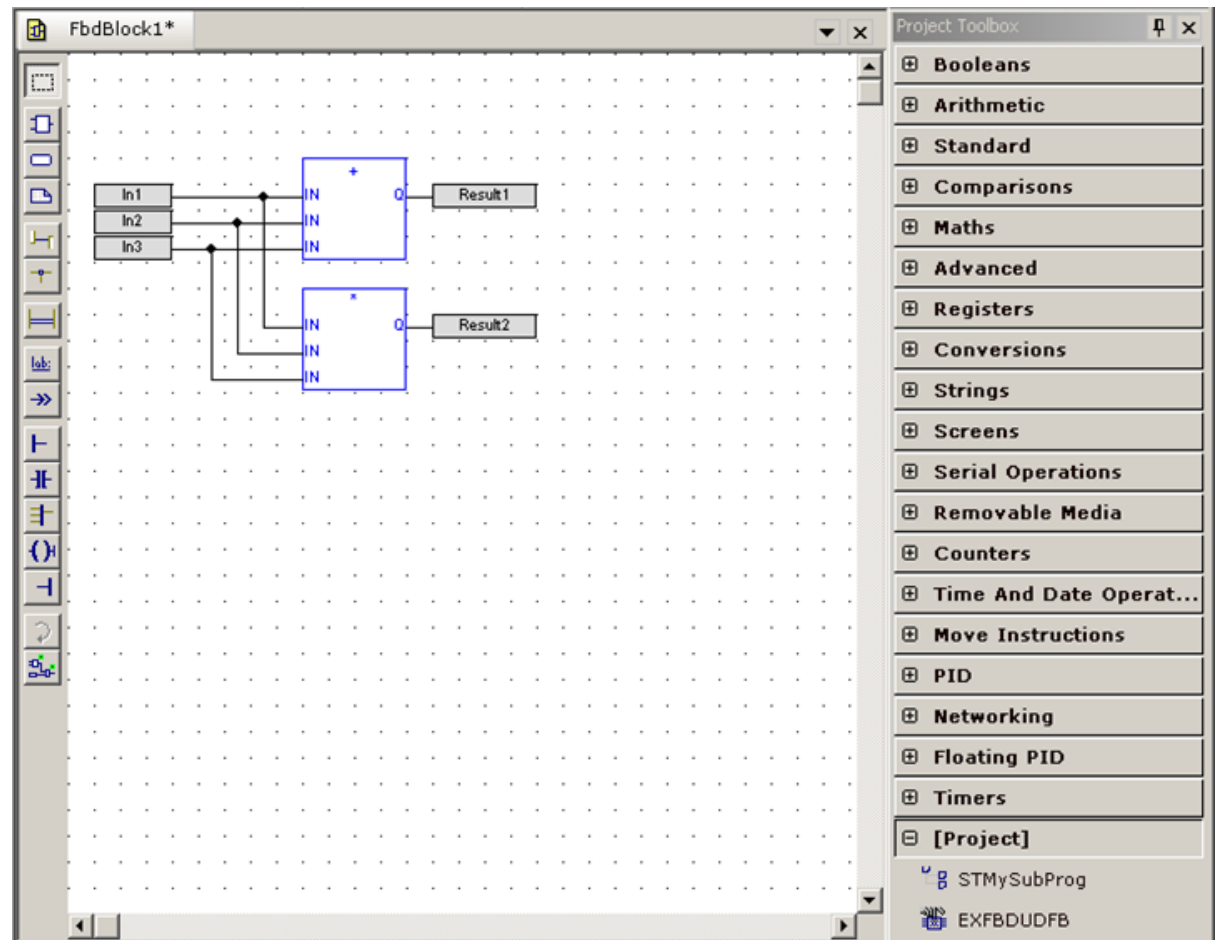
### Closing SFC secondary editors

The close icon on the secondary editor can be used to close individual steps/transitions. The tab right click options listed above can also be used for multiple tab closures.

## FBD Editor

### Function Block Diagram (FBD) Editor

The FBD editor is a powerful graphical tool that enables you to enter and manages Function Block Diagrams according to the IEC 61131-3 standard. The editor supports advanced graphic features such as drag and drop, object resizing and connection lines routing features, so that you can rapidly and freely arrange the elements of your diagram. It also enables you to insert in a FBD diagram graphic elements of the LD (Ladder Diagram) language such as contacts and coils.



*FBD diagram components:*

Function  
Variable  
Comment  
Corners  
Network  
Labels  
Jumps

*Related sections:*

Using the FBD toolbar  
Selecting function blocks  
Drawing connection lines  
tags  
Selecting and entering variables and FB instances  
texts  
Viewing the diagram  
Moving or copying parts of the diagram  
breaks  
Inserting an object on a line  
Resizing objects  
Modifying Function blocks  
Bookmarks

*LD components:*

Contacts

Coils

"OR"

Power rails

vertical

rail















Note: When a contact or a coil is selected, You can press the SPACE bar to change its type (normal, negated, pulse...).



## Using the FBD Toolbar

### Using the FBD Toolbar

The vertical toolbar on the left side of the FBD editor contains buttons for all available editing features.

-  **Element Selection:** In this mode, no insertion of any element possible in the diagram. The mouse is used for selecting object and lines, select tag name areas, move or copy objects in the diagram. At any moment user can press the ESCAPE key to go back to the Selection mode.
-  **Add function Block:** In this mode, the mouse is used for inserting blocks in the diagram. Click in the diagram and drag the new block to the wished position. The type of block that is inserted is the one currently selected in the list of function blocks.
-  **Add variable:** In this mode, the mouse is used for inserting variable tags. Variable tags can then be wired to the input and output pins of the blocks. Click on the diagram and drag the new variable to the wished position.
-  **Add comment:** In this mode, the mouse is used for inserting comment text areas in the diagram. Comment texts can be entered anywhere. Click in the diagram and drag the text block to the wished position. The text area can then be selected and resized.
-  **Add arc:** In this mode, the mouse is used to wire input and output pins of the diagram objects. The line must always be drawn in the direction of the data flow: from an output pin to an input pin
-  **Add Corner:** In this mode, the mouse is used for inserting a user defined corner on a line.
-  **Add break:** In this mode, the mouse is used for inserting a horizontal line that acts as a break in the diagram. Breaks have no meaning for the execution of the program. They just help the understanding of big diagrams, by splitting them in a list of networks.
-  **Add label:** In this mode, the mouse is used for inserting a label in the diagram. A label is used as a destination for jump symbols.
-  **Add jump:** In this mode, the mouse is used for inserting jump symbols in the diagram. A jump indicates that the execution must be directed to the corresponding label (having the same name as the jump symbol). Jumps are conditional instructions. They must be linked on their left side to a Boolean data flow.
-  **Add left power rail:** In this mode, the mouse is used for inserting a left power rail in the diagram. A left power rail is an element of the LD language, and represents a "TRUE" state that can be used to initiate a data flow. Power rails can then be selected and resized vertically according to the wished network height.
-  **Add direct contact:** In this mode, the mouse is used for inserting in the diagram a contact as in Ladder Diagrams.
-  **Add "OR" Bar:** In this mode, the mouse is used for inserting a rail that collects several Boolean data flows for an "OR" operation, in order to insert parallel contacts such as done in Ladder Diagrams.
-  **Add direct coil:** In this mode, the mouse is used for inserting in the diagram a coil as in Ladder Diagrams. It is not mandatory that a coil be connected on its right side.
-  **Add right power rail:** In this mode, the mouse is used for inserting a right power rail in the diagram. A right power rail is an element of the LD language, and is commonly used for terminating Boolean data flows. However it is not mandatory to connect coils to power rails. Right power rails have no meaning for the execution of the diagram.



Swap item style: Swaps visibility between tag & description for a variable. Space bar can also be used for this function.



Show execution order: Shows execution order for blocks.

## FBD Variables

All variable symbols and constant expressions are entered in FBD diagrams using small boxes. Press the following button in the FBD toolbar for inserting a variable tag:



**Insert variable:** In this mode, the mouse is used for inserting variable tags. Click in the diagram and drag the new variable to the wished position.

Double click on a variable tag to open the variable selection box and either select the symbol of the wished variable or enter a constant expression.

Variables tags must then be linked to other objects such as block inputs and outputs using connection lines.

You can resize a variable box vertically in order to display together with the variable name its tag (explicit link to the registers), its description text. The variable name is always displayed at the bottom of the rectangle:

description
tag
name

## FBD Comments

Comment text areas can be entered anywhere in a FDB diagram. Press the following button in the FBD toolbar for inserting a new comment area:



**Insert comment text:** In this mode, the mouse is used for inserting comment text areas in the diagram. Comment texts can be entered anywhere. Click in the diagram and drag the text block to the wished position.

Double click on the comment area for entering or changing the attached text. When selected, comment texts can be resized.

## FBD Corners

Corners are used to force the routing of connection lines, as the FBD editor imposes a default routing only between two pins or user defined corners. All variable symbols and constant expressions are entered in FBD diagrams using small boxes. Press the following button in the FBD toolbar for inserting a corner on a line:



**Insert corner:** In this mode, the mouse is used for inserting a user defined corner on a line.

You can drag a new line from an output pin to an empty space. In that case the editor automatically finished the line with a user defined corner so that you can continue drawing the connection to the wished pin and force the routing while you are drawing the line.

Corners can then be selected and moved to change the routing of existing lines.

### FBD Network Breaks

Network breaks can be entered anywhere in a FBD diagram. Breaks have no meaning for the execution of the program. They just help the understanding of big diagrams, by splitting them in a list of networks. Press the following button in the FBD toolbar for inserting a new break:




**Insert network break:** In this mode, the mouse is used for inserting a horizontal line that acts as a break in the diagram.

The break line is drawn on the whole diagram width. No other object can overlap a network break. Break lines can then be selected and moved vertically to another location.

Network breaks can also be used for browsing the diagram. Press **Ctrl+Page Up** or **Ctrl+Page Down** keys to move the selection to the next or previous network break.


### FBD "OR" Vertical Rail

The FBD editor enables the drawing of LD rungs. A particular object, the "OR" rail can be inserted on a rung in order to connect parallel contacts together press the following button in the FBD toolbar for inserting a new "OR" rail:

 **Insert "OR" rail:** In this mode, the mouse is used for inserting a rail that collects several Boolean data flows for an "OR" operation, in order to insert parallel contacts such as done in Ladder Diagrams.

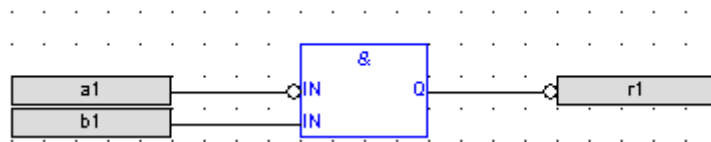
The "OR" rail has exactly the same meaning as an "OR" block regarding the execution of the diagram.

### Drawing FBD Connection Lines


 Press this button before inserting a new line.


The editor enables you to terminate a connection line with a boolean negation represented by a small circle. To set or remove the boolean negation, select the line and press the SPACE bar.


### Example:





Connection lines must always be drawn in the direction of the data flow: from an output pin to an input pin. The FBD editor automatically selects the best routing for the new line. Connection lines indicate a data flow between the following possible objects:


 **Block:** Refer to the help on the block for the description of its input and output pins, and the expected data types for the coherency of the diagram.


 **Variable:** Variables can be connected on their right side (to initiate a flow) or on their left side for forcing the variable, if it is not "read only". The flow must fit the data type of the variable.


 **Jump:** a jump must be connected on its left side to a Boolean data flow.

 **Left power rail:** Left power rails represent a TRUE state and can be connected to a non limited number of objects on their right side.

 **Contact:** A contact must be connected on its left side and on its right side to Boolean data flows.

 **"OR" rail:** Such rail that collects several Boolean data flows for an "OR" operation, in order to insert parallel contacts such as done in Ladder Diagrams. It may have several connections on its left side and on its right side. All connected data flows must be Boolean.

 **Coil:** A coil must be connected on its left side to a Boolean data flow. It is not mandatory that a coil be connected on its right side.


 **Right power rail:** A right power rail is an element of the LD language, and is commonly used for terminating Boolean data flows. It has a non limited number of connections on its left side. It is not mandatory to connect coils to power rails.





### Selecting FBD Variables and Instances


 *Press this button or press ESCAPE before any selection.*


To select the name of the declared variable to be attached to a graphic symbol, you must be in "Selection" mode. Simply double click on the tag name gray area. The following types of object must be linked to valid symbols:


 **Block:** If it is a function block, you must specify the name of a valid declared instance of the corresponding type. The Function blocks can be selected by double-clicking the blocks and selecting the required block from the list of blocks available. The number of operands can also be selected.

 **Variable:** This field must be attached to a declared variable. Also, a variable box may contain the text of a valid constant expression. Variables will have the names of the registers of operands.

 **Label:** This function is used to break the execution of the program and jump to a desired place in the program. It must have a unique name within the diagram. This operation would be the destination for the corresponding Jump operation.

 **Jump:** This function is used to break the execution of the program and jump to a desired place in the program. It must have the same name as declared using the destination Label operation within the diagram.

 **Contact:** This must be attached to a declared Boolean variable or after placing this symbol on the diagram a Boolean variable should be declared as a contact.

 **Coil:** This must be attached to a declared Boolean variable or after placing this symbol on the diagram a Boolean variable should be declared as a coil.

Symbols of variables and instances are selected using the variable list, that can be used as the variable editor.

You can simply enter a symbol or constant expression in the edit box and press OK. You also can select a name in the list of declared object, or declare a new variable by pressing the "Create" button.

more details...

**Viewing FBD Diagrams**

The diagram is entered in a logical grid. All objects are snapped to the grid.

At any moment you can use the View | Zoom menu option for zooming in or out the edited diagram. You also can press the [+] and [-] keys of the numerical keypad for zooming the diagram in or out.

## Moving or Copying FBD Objects



*Press this button or press ESCAPE before selecting objects.*

The FBD editor fully supports drag and drop for moving or copying objects. To move objects, select them and simply drag them to the wished position.

To copy objects, you may do the same, and just press the CONTROL key while dragging. It is also possible to drag pieces of diagrams from a program to another if both are open and visible on the screen.

At any moment while dragging objects you can press ESCAPE to cancel the operation.

Alternatively, you can use classical Copy / Cut / Paste commands from the Edit menu. When you run the Paste command, the editors turns in "Paste" mode, with a special mouse cursor. Click in the diagram and move the mouse cursor to the wished position for inserting pasted objects.

### Using the Keyboard

When graphic objects are selected, you can move them in the diagram by hitting the following keys:

Shift + Up	Move to the top
Shift + Down	Move to the bottom
Shift + Left	Move to left
Shift + Right	Move to right

When an object is selected, you can extend the selection by hitting the following keys:

Shift + Control + Home	Extend to the top: select all objects before the selected one
Shift + Control + End	Extend to the bottom: select all objects after the selected one

To insert or delete space in the diagram, you can simply select an object, press Shift+Control+End to extend the selection and then move selected objects up or down.

### Auto Alignment

When objects are selected, the following keystrokes automatically align them:

Control + Up	To the top
Control + Down	To the bottom
Control + Left	To left
Control + Right	To right

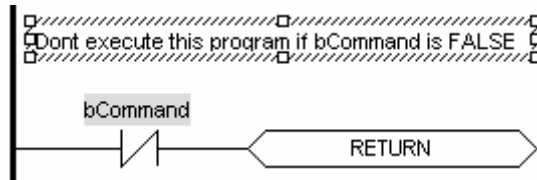
**Inserting FBD Objects on a Line**

The FBD editor enables you to insert an object on an existing line and automatically connect it to the line. This feature is available for all objects having one input pin and one output pin, such as variable boxes, contacts and coils. This feature is mainly useful when entering pieces of Ladder Diagrams. Just draw a horizontal line between left and right power rails: this is the rung. Then you can simply insert contacts and coils on the line to build the LD rung.

## Resizing FBD Objects

 Press this button or press *ESCAPE* before selecting objects.

When an object is selected, small square boxes indicates you how to resize it with the mouse. Click on the small square boxes for resizing the object in the wished direction.



Not all objects can be resized. The following table indicates possible operations:

Variable	Horizontally and vertically (*)
Block	Horizontally
Labels and jumps	Horizontally
Power rails	Vertically (Before connecting elements to them)
OR rail	Vertically (Before connecting elements to them)
Comment area	In all directions

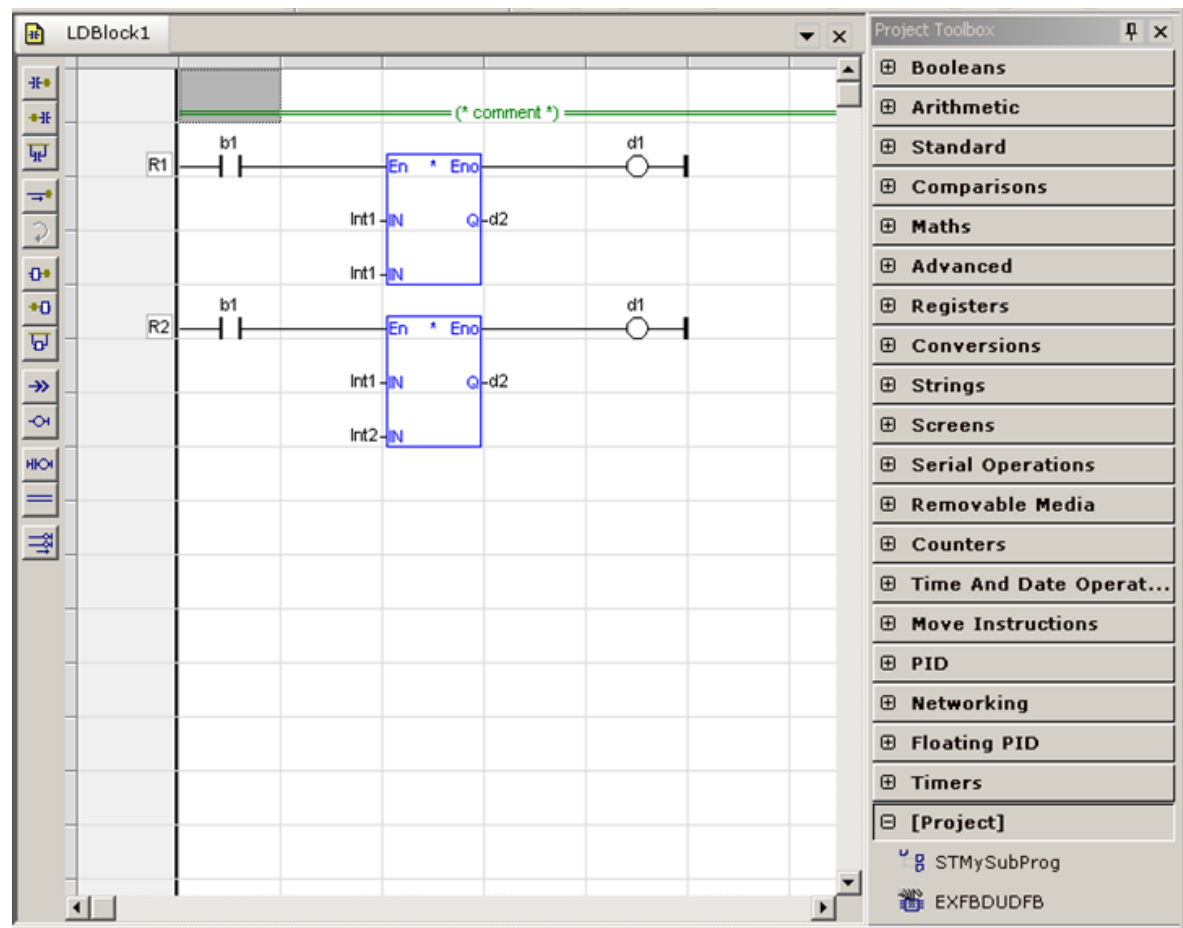
(\*) Resizing a variable box vertically enables you to display together with the variable name its tag (short comment text), its description text, plus its I/O location if the variable is mapped to an I/O channel. The variable name is always displayed at the bottom of the rectangle:

description
tag
name

## LD Editor

### Ladder Diagram (LD) Editor

The LD editor is a powerful graphical tool that enables you to enter and manage Ladder Diagrams according to the IEC 61131-3 standard. The editor enables quick input using the keyboard, and supports advanced graphic features such as drag and drop.



#### LD diagram components:

Rungs  
Contacts  
Coils  
Power  
Function  
Labels  
Jumps  
Comments














#### Related sections:

Using the LD toolbar  
Selecting function blocks  
Selecting and entering variables and FB instances  
Viewing the diagram  
Moving or copying parts of the diagram  
Modifying Function blocks  
Bookmarks

**Note:** When a contact or a coil is selected, You can press the SPACE bar to change its type (normal, negated, pulse...)

### Using the LD Toolbar

The vertical toolbar on the left side of the editor contains buttons for inserting items in the diagrams. Items are inserted at the current position in the diagram.

	Shift+F4	Insert a contact before the selected item.
	F4	Insert a contact after the selected item.
	Ctrl+F4	Insert a contact in parallel with the selected items
	Ctrl+Space	Insert a horizontal line before the selected item so that it is pushed to the right.
	Mouse Click	Swap elements between NO, NC, Positive triggered or Negative triggered for Contacts and Coils. Additionally swap Set and Reset Elements for Coils. Space bar can also be used for this function.
	Shift+F8	Insert a block before the selected item.
	F8	Insert a block after the selected item.
	Ctrl+F8	Insert a block in parallel with the selected items.
	Shift+F9	Add a jump in parallel with the selected coil.
	F9	Add a coil in parallel with the selected coil.
	Ctrl+R	Insert a new rung in the diagram.
	Ctrl+D	Insert a comment between rungs.
	Mouse Click	Align Coils to the right to uniformly display the networks.

## Managing Rungs



**Ctrl+R:** Press this button in the LD toolbar to insert a new rung.

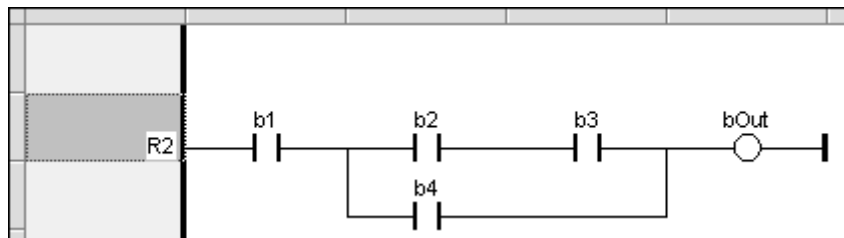


**Shift+F4:** Insert a contact before the selected item. This icon also adds a rung when the Rung is blank

A LD diagram is a sequential list of rungs. Each rung represents left to right boolean power flow, that begins with a power rail, always drawn in the first column of the diagram, and finishes with a coil or a jump symbol.

Each Rung is identified by a default numbered identifier (*Rnnn*) displayed on the left of the power rail. The rung identifier can be used as a target for jump instructions. Alternatively you can enter a specific rung label by double clicking in the rung head on the left margin.

The LD editor enables you to manipulate whole rungs by selecting only their head in the left margin. The following example shows a selected rung:



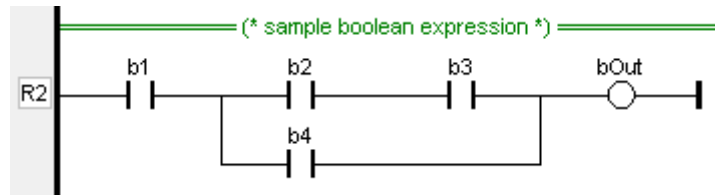
When a rung is selected, use the right click options to delete, copy or cut it.



### Comments in LD Diagrams

 Press this button in the LD toolbar to insert a new comment line.

The LD editor enables you to insert comment texts in the diagram. A comment is a single line of text inserted between two rungs. The comment text is displayed on a double line in the diagram:



Comment texts have no meaning for the execution of the diagram. They are used to enhance the readability of the program, enabling the description of each rung.

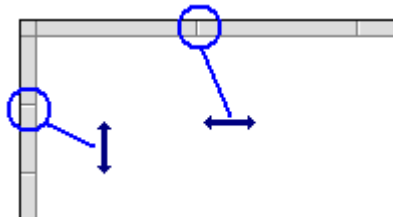
The comment text remains visible when the diagram is scrolled horizontally. To change the text of the comment, place the selection anywhere on the comment line and hit ENTER key, or simply double click on the comment line.

### Viewing LD Diagrams

The diagram is entered in a logical grid. All objects are snapped to the grid.

At any moment you can use the commands of the "View" menu for zooming in or out the edited diagram. You also can press the [+] and [-] keys of the numerical keypad for zooming the diagram in or out.

You also can drag the separation lines in vertical and horizontal rulers to freely resize the cells of the grid:



The LD editor adjust the size of the font according to the zoom ratio so that the name of variables associated with contacts and coils are always visible. If cells have sufficient height, variable names are completed with other pieces of information about the variable:

- its tag (association with the target registers)
- its description text

### **Moving and Copying LD Items**

The LD editor fully supports drag and drop for moving or copying objects. To move objects, select them and simply drag them to the wished position, in the same rung or in another rung.

To copy objects, you may do the same, and just press the CONTROL key while dragging. It is also possible to drag pieces of diagrams from a program to another if both are open and visible on the screen.

At any moment while dragging objects you can press ESCAPE to cancel the operation.

Alternatively, you can use classical Copy / Cut / Paste commands from the Edit menu. Paste is performed at the current position.

You can manipulate whole rungs by selecting only their head in the left margin (select only the cell where the rung number is displayed).

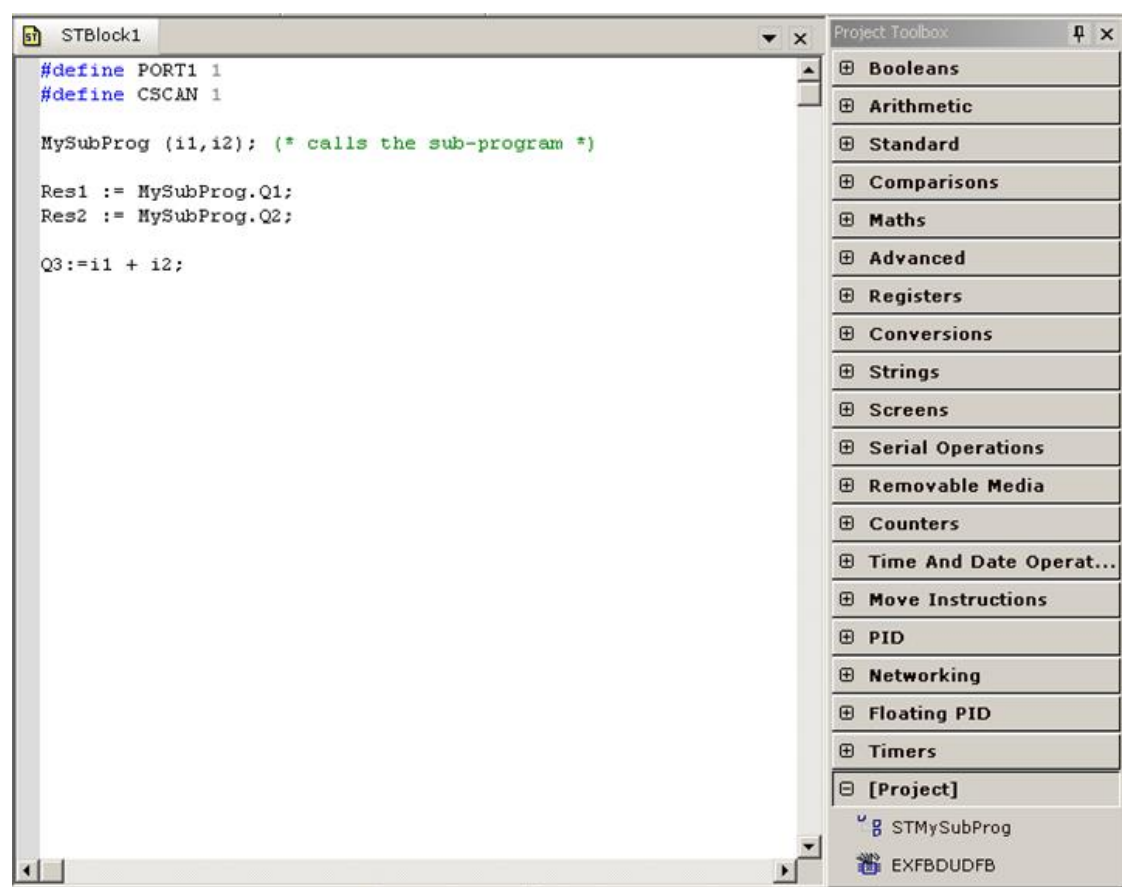
## ST Editor

### Structured Text (ST) Editor

The ST editor is a powerful language sensitive text editor dedicated to IEC 61131-3 languages. The editor supports advanced graphic features such as drag and drop, syntax coloring and active tooltips for efficient input and test of programs in ST.

ST is a structured literal programming language. A ST program is a list of statements. Each statement describes an action and must end with a semi-colon (";").

The presentation of the text has no meaning for a ST program. You can insert blank characters and line breaks where you want in the program text.

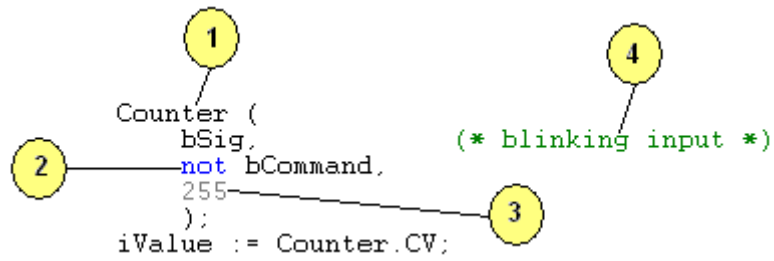


*Related sections:*

- Syntax coloring
- Auto-completion of words
- Drag and drop
- Active tooltips

### ST / IL Syntax Coloring

The ST / IL editor supports syntax coloring according to the selected programming language (ST or IL). The editor uses different colors for the following kind of words:



1. Default (identifiers, separators...)
2. Reserved keywords of the language
3. Constant expressions
4. Comments

### Auto Completion of Words

The ST / IL editor includes powerful commands for automatic completion of typed words, according to declared variables and data types. The following features are available:

#### Auto completion of a variable name

If you enter the first letters of a variable name, you can hit the **CTRL+SPACE** keys for automatically completing the name. A popup list is displayed with possible choices if several declared variable names match the type characters.

#### Selection of FB member

When you type the name of a function block instance (use either as an instance or a data structure), pressing the point "." after the name of the instance opens a popup list with the names of possible members.

#### Other syntax related commands

When lines are selected, you can automatically indent them. Press **TAB** or **Shift+TAB** keys to shift the lines to the left or to the right, by adding or removing blank characters on the left.

### **ST / IL Drag and Drop Features**

The ST / IL editor supports powerful drag and drop features that help you developing and testing your programs. You can:

- drag text (words or lines) from the ST / IL editor to another application (such as a text editor)
- do the opposite
- drag a variable symbol from the Program Variables Window to the ST / IL editor
- drag a variable symbol from the ST / IL editor to the watch list (\*)

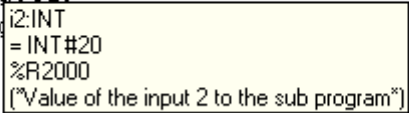
(\*) When dragging the symbol of an array to the watch list, all items of the array are added to the watch list.

### Tooltips in the ST Editor

You don't need to run any specific command to open the tooltip. Just activate the editor window by clicking upon it and put the mouse on the variable symbol and wait for one second. The tooltip will show fields of the variable pointed to by the mouse cursor. However, the Tool tip has to be enabled in the IEC Language Editor settings window and the specific fields need to be selected. It can be accessed from Menu-Tools/Editor options.

In the Debug mode, the tool tip will show the current value in addition to all the fields shown in Editor mode.

```
#define PORT1 1
#define CSCAN 1
MySubProg (i1, i2); (* calls the sub-program *)
Res1 := MySubProg.Q1;
Res2 := MySubProg
```



The tooltip displays the following information:

- i2:INT
- = INT#20
- %R2000
- ("Value of the input 2 to the sub program")

The value shown in the tooltip is automatically refreshed while the tooltip is open.

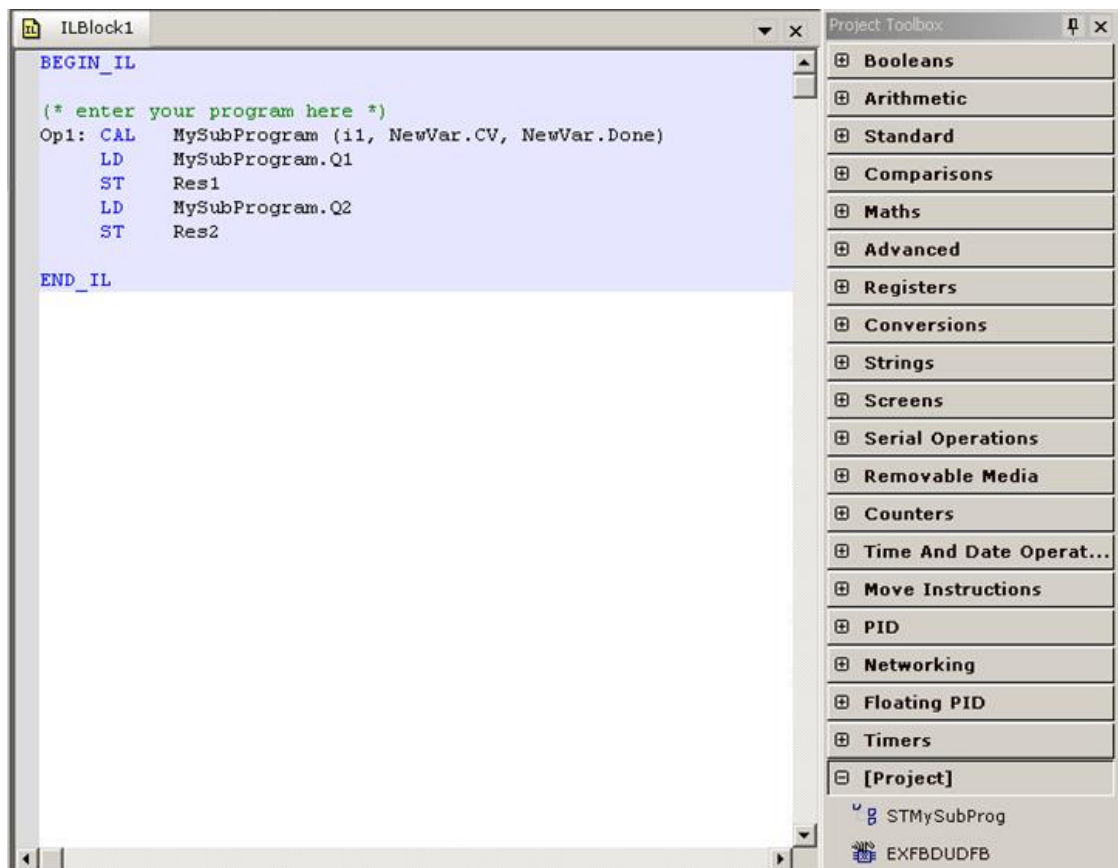


## IL Editor

### Instruction List (IL) Editor

The IL editor is a powerful language sensitive text editor dedicated to IEC 61131-3 languages. The editor supports advanced graphic features such as drag and drop, syntax coloring and active tooltips for efficient input and test of programs in IL.

A program written in IL language is a list of instructions. Each instruction is written on one line of text. An instruction may have one or more operands. Operands are variables or constant expressions. Each instruction may begin with a label, followed by the ":" character. Labels are used as destination for jump instructions.



IL instructions in the program must be entered between BEGIN\_IL and END\_IL keywords, such as in the following example:

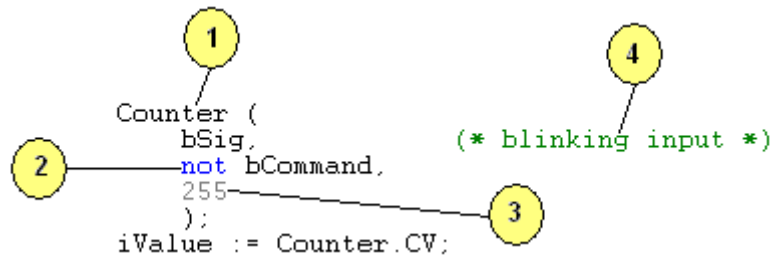
```
BEGIN_IL
(* enter your program here *)
END_IL
```

*Related sections:*

Syntax coloring  
Auto-completion of words  
Drag and drop  
Active tooltips  
Selecting function blocks  
Inserting variable and FB instances symbols  
Bookmarks

### ST / IL Syntax Coloring

The ST / IL editor supports syntax coloring according to the selected programming language (ST or IL). The editor uses different colors for the following kind of words:



1. Default (identifiers, separators...)
2. Reserved keywords of the language
3. Constant expressions
4. Comments

### Auto Completion of Words

The ST / IL editor includes powerful commands for automatic completion of typed words, according to declared variables and data types. The following features are available:

#### Auto completion of a variable name

If you enter the first letters of a variable name, you can hit the **CTRL+SPACE** keys for automatically completing the name. A popup list is displayed with possible choices if several declared variable names match the type characters.

#### Selection of FB member

When you type the name of a function block instance (use either as an instance or a data structure), pressing the point "." after the name of the instance opens a popup list with the names of possible members.

#### Other syntax related commands

When lines are selected, you can automatically indent them. Press **TAB** or **Shift+TAB** keys to shift the lines to the left or to the right, by adding or removing blank characters on the left.

### **ST / IL Drag and Drop Features**

The ST / IL editor supports powerful drag and drop features that help you developing and testing your programs. You can:

- drag text (words or lines) from the ST / IL editor to another application (such as a text editor)
- do the opposite
- drag a variable symbol from the Program Variables Window to the ST / IL editor
- drag a variable symbol from the ST / IL editor to the watch list (\*)

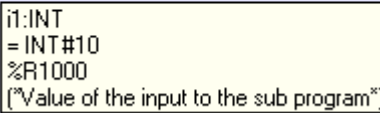
(\*) When dragging the symbol of an array to the watch list, all items of the array are added to the watch list.

### Tooltips in the IL Editor

You don't need to run any specific command to open the tooltip. Just activate the editor window by clicking upon it and put the mouse on the variable symbol and wait for one second. The tooltip will show fields of the variable pointed to by the mouse cursor. However, the Tool tip has to be enabled in the IEC Language Editor settings window and the specific fields need to be selected. It can be accessed from Menu-Tools/Editor options.

In the Debug mode, the tool tip will show the current value in addition to all the fields shown in Editor mode.

```
BEGIN_IL
(* enter your program here *)
Op1: CAL    MySubProgram (i1, i2)
      LD     MySubProgram.Q1
      ST     Res1
      LD     MySubProgram.Q2
      ST     Res2
END_IL
```

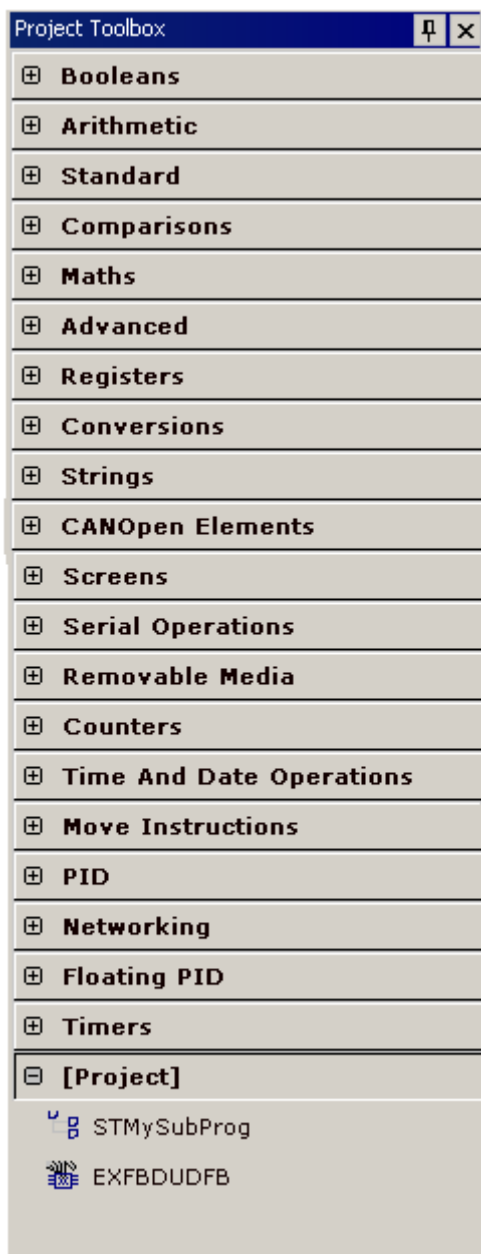


The value shown in the tooltip is automatically refreshed while the tooltip is open.

## Selecting Function Blocks Using IEC Project Toolbox

### Project Toolbox

The Project Toolbox simplifies adding of logic blocks to the logic editing area. The blocks are grouped according to their functionality.



Each group can be expanded independently to access the various functions.

Left clicking the function and dragging it to the logic editing area, places the element there.

If the project has subroutines or UDFBs defined, the same will also be listed under 'Project' in toolbox for easy drag and drop.

Notes:

**The elements that are not supported by the configured model will be grayed.**

**The Project Toolbox contents are not applicable when editing the main SFC Module in IEC Programs.**

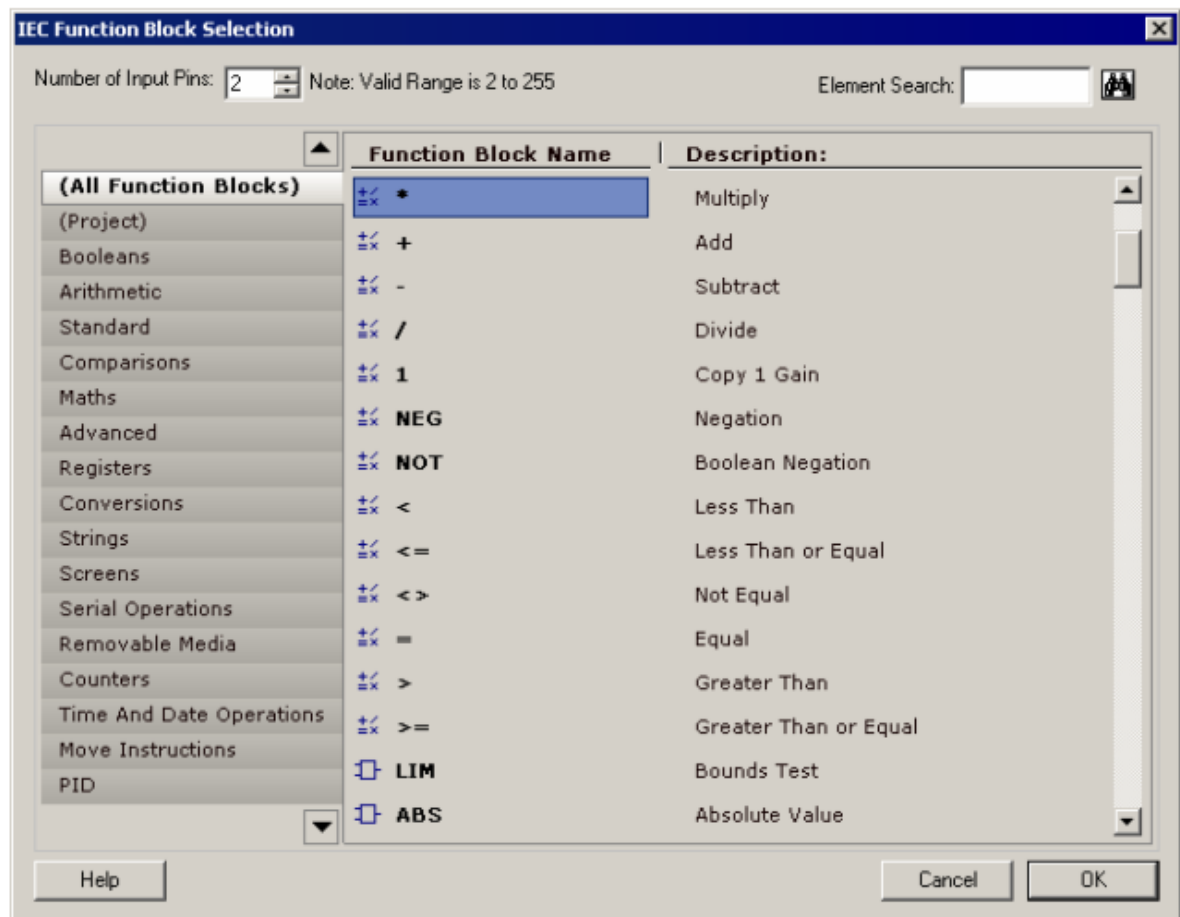
**In this case, the toolbox window contents will be empty.**



## Modifying Function blocks

IEC Project Toolbox

Double-clicking on the Function blocks open the IEC Function blocks selection window which lists out all the function blocks present for the IEC Editor. This allows user to change an existing function block to a different one.



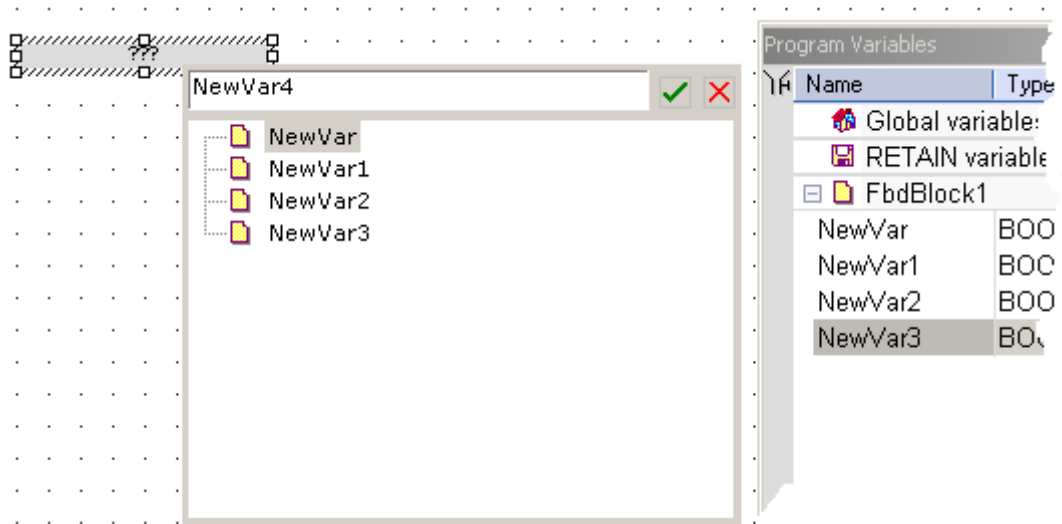
This dialog also allows to change the number of inputs for few blocks. Eg. the number of inputs for Add block by default are 2 but can be changed by changing 'Number of Input Pins' in this dialog.

Note: For blocks which do not allow variable number of inputs, this edit box is greyed.

This feature is available for FBD and LD language editors only.

## Selecting Variables and Instances

Symbols of variables and instances are selected using the variable list, that can be used as the variable editor. Selecting variables is available from all editors:



- In FBD diagrams, double click on a variable box, a FB instance name, a contact or a coil to select the associated variable.
- In LD diagrams, double click on a contact, a coil or a block input or output to select the variable. Double click on the top of a FB rectangle to select an instance.
- In ST/IL texts, place the cursor where variable needs to be inserted. Right click and select Insert/Set Variable.

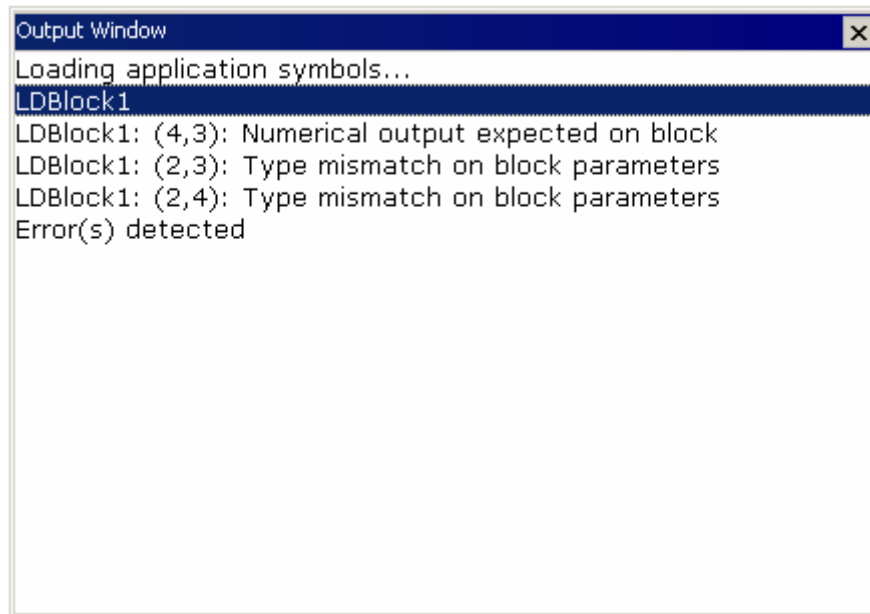
When the Program Variables Window is enabled from **Tools Menu**, you can simply drag a variable from the list to the program to insert it.

If Editor Options > Diagrams > 'Prompt for Variable Name after insert' is checked, the Program Variables dialog gets opened when a variable is placed in the logic editing area.

When inserting a variable name manually, the Workbench automatically checks if it exists and if not, proposes you to declare it right now, if 'Ask for Declaration when new variable is inserted' is enabled in Editor Options > Diagrams.

## **Output Window**

The Output Window is a dockable window which lists all the compilation errors in an application. Double clicking on the error in the list, goes to the error location without closing the output window.



Like all dockable windows, it can be placed in pinned, hidden, docking and floating mode anywhere in the working area.

### **Output Window Details:**

The output window lists out the errors present in each of the blocks by checking block by block.

The Error List uses the following Syntax:

[BlockType and No.]: [Location – Co-ordinates of the error]: [Description]

[BlockType and No.]: Displays the type of block that has reported the error and the no. assigned to it in case of more than one block of same type.

[Location – Co-ordinates of the error]: Gives the Row and Column location of the offending element.

[Description] gives a short description of the problem

## Defines

### Definitions

Definitions are typically used for replacing a constant expression and ease the maintenance of programs.

There are three levels of definitions:

- Pre-defined Library of Defines
- Global to all program blocks of a project
- Local to one program block

Each definition must be entered on one line of text according to the following syntax:

```
#define Identifier Equivalence (* comments *)
```

Below are some examples:

```
#define OFF FALSE (* redefinition of FALSE constant *)  
#define PI 3.14 (* numerical constant *)  
#define ALARM (bLevel > 100) (* complex expression *)
```

You can use a definition within the contents of another definition. The definition used in the other one must be declared first. Below is an example:

```
#define PI 3.14  
#define TWOPI (PI * 2.0)
```

Notes:

A definition may be empty, for example:

```
#define CONDITION
```

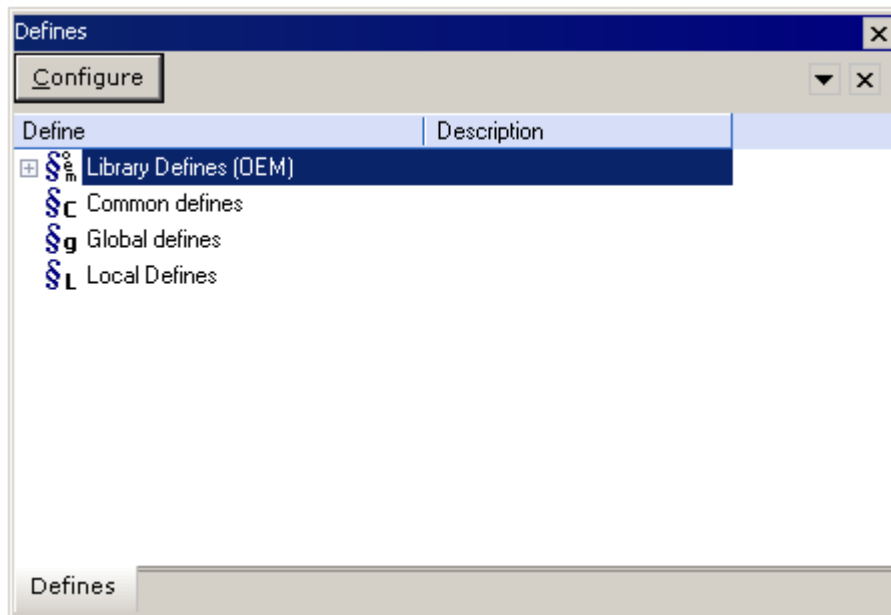
Defined word can be used for directing the conditional compiling directives.

You can enter #define lines directly in the source code of programs in IL or ST languages.

The use of definitions may disturb the program monitoring and make error reports more complex. It is recommended to restrict the use of definitions to simple expressions that have no risk to lead to a misunderstanding when reading or debugging a program.

## Defines Window

The Defines Window is a window where the constant values can be assigned.



It includes the Pre-defined system constants known as Library Defines (OEM) as well as the Global defines which can be used in all the blocks and Local defines which are block specific.

The Contents of a particular section of the defines editor can be edited by double-clicking that section or by selecting it and clicking the Configure button to bring up the Defines Editor Tabbed Window in the editing area.

A Defines Editor is opened for the Global Defines and for the block specific Local Defines where the variables can be viewed/edited in the logic editing and assigned a constant value.

## Bookmarks

Bookmarks are used for navigating in a document. You can freely insert bookmarks everywhere in a document. Then you can jump from a bookmark to another with a single command for browsing the document. Bookmarks are supported in all program editors plus the Program Variables Window.

Below are the available commands for using bookmarks:

<b>Ctrl + F2</b>	Toggle the bookmark at the current position
<b>Shift + F2</b>	Go to the next bookmark

Depending on the type of document, the possible locations for a bookmark are:


- In the text editor, a bookmark is placed on a line of text.
- In the SFC editor, a bookmark is placed on a SFC symbol (step, transition, jump...).
- In the FBD editor, a bookmark is placed on any FBD object (not on a line).
- In the LD editor, a bookmark is placed on a rung header.
- In the Program Variables Window, a bookmark is placed on any line of the grid (variable or group).

**Note:** Bookmarks are valid only while the editing window is open, and are not stored in the document when the window is closed.

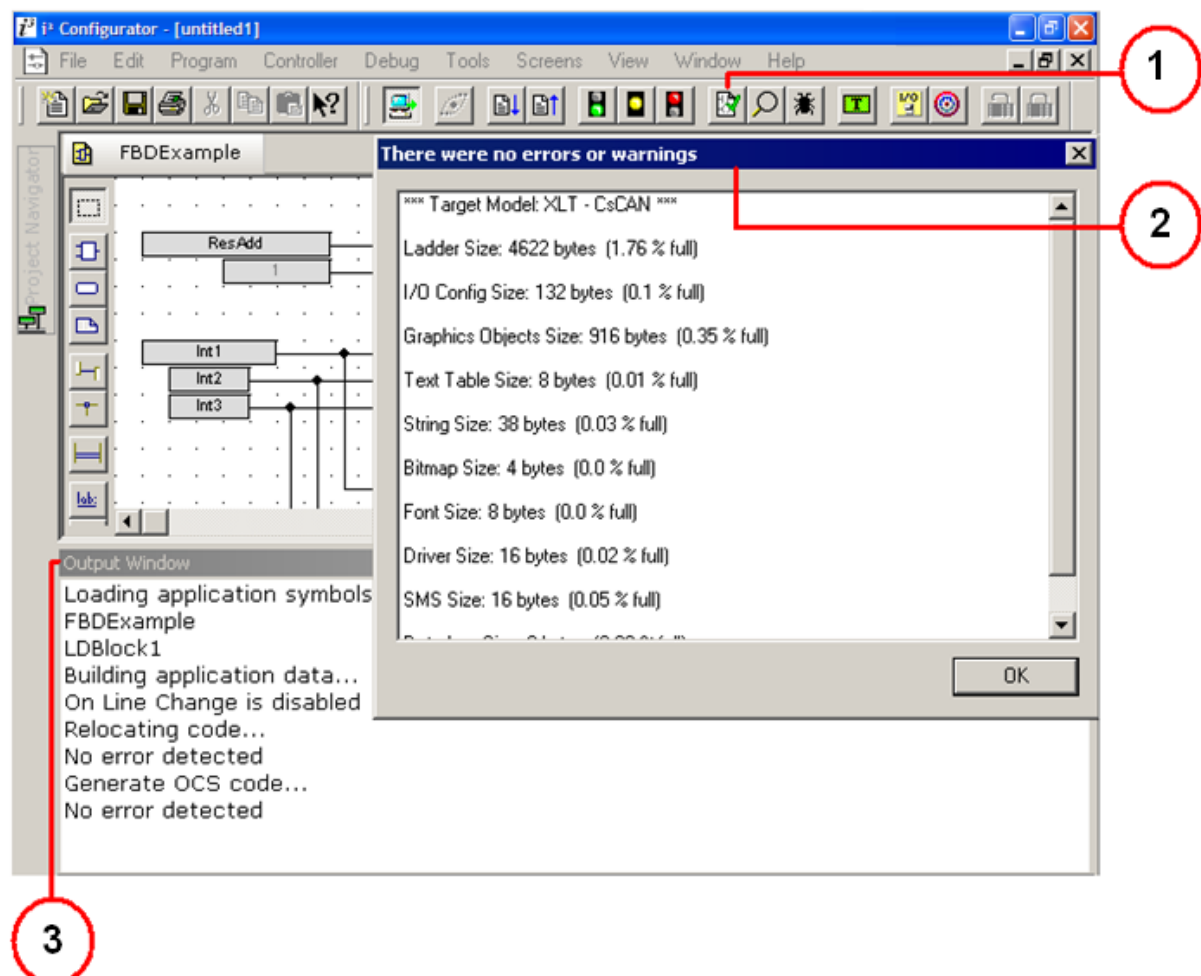
## Check a Program for Errors in IEC Programs

Program Errors are problems present in the Language blocks that prevent the program from running, such as unconnected elements, data type mismatch for operands, etc; ERRORS must be corrected before the program can be downloaded.

IEC Programs are automatically compiled for syntactical errors before they are downloaded to the controller. The Output window is displayed with the results of the compilation.

Manually check a program for errors by selecting Program | Error Check from the Main Menu or selecting the Error Check tool  from the Unit communications Toolbar.

If there are no errors, then a No-Error Message box appears, along with details in the Output Window:



1) Error Check Icon: This Icon from the Unit communications tool bar helps in manual compilation of the program.

2) No-Error Message Box: This contains the details of the program in terms of memory usage by various functions.

3) Output Window: Lists out the details of the Errors present in the program.

If there are Errors present in the program then the Error checking results are displayed only in the docking Output window. Clicking an error location in error list goes to the error location without closing the error window.

The output window lists out the errors present in each of the blocks by checking block by block.

The Error List uses the following Syntax:

[Modulename]: [Location – Co-ordinates of the error]: [Description]

[Modulename]: Displays the type of block that has reported the error and the no. assigned to it in case of more than one block of same type.

[Location – Co-ordinates of the error]: Gives the Row and Column location of the offending element.

[Description] gives a short description of the problem



## Debugging the Application

### *Debugging the Application*

Use the following commands in the main window to test or monitor your application:



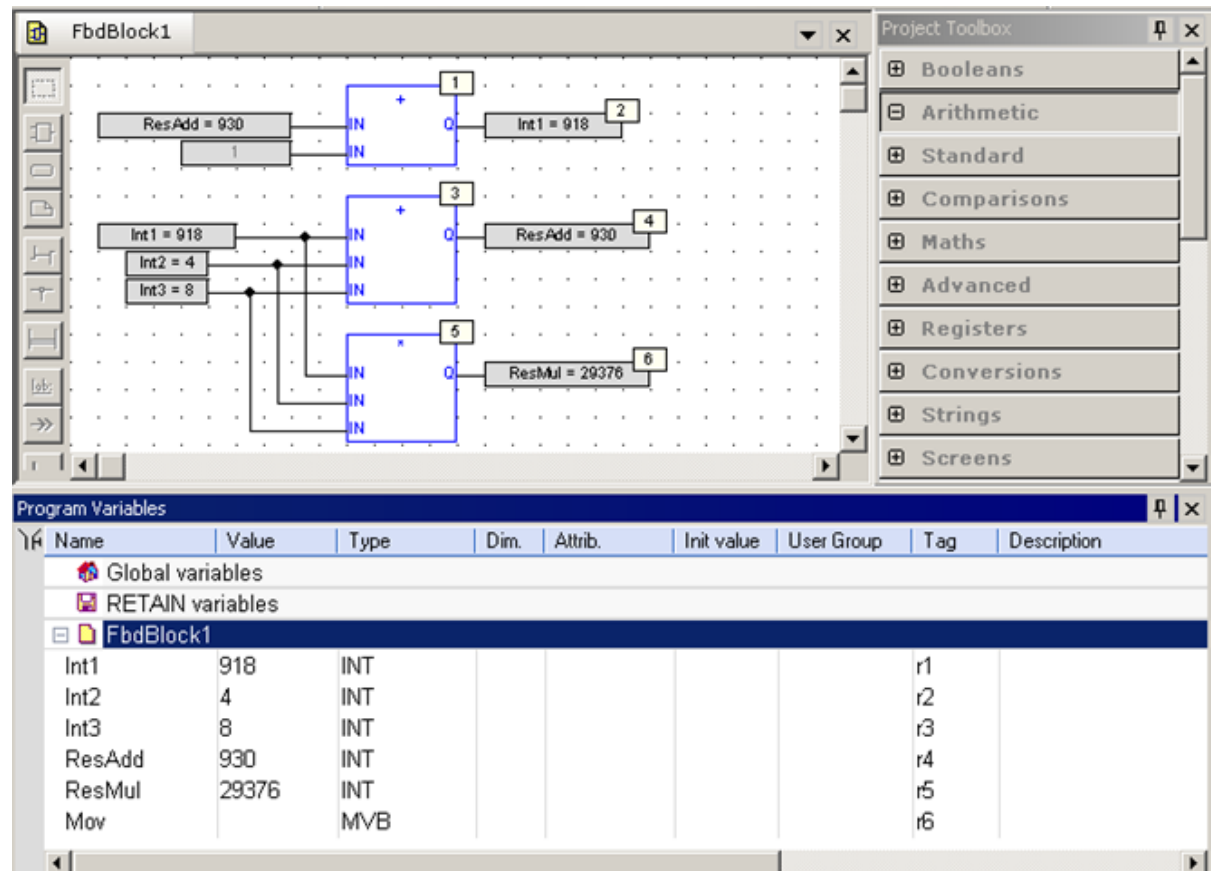
Press this button to go On Line/Debug mode and establish the connection with the remote target. Project monitoring will be available when the connection is established.

How to monitor the application:

- Using the editors in Debug mode
- Using the Watch Window

## Using the Editors in Debug Mode

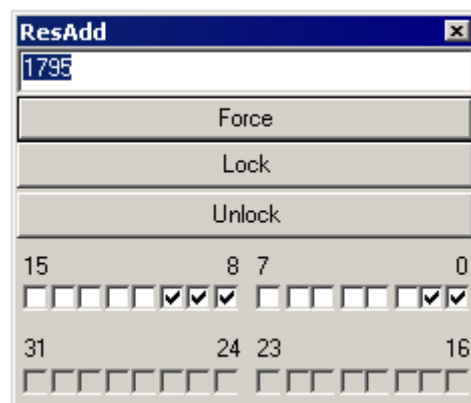
In Debug mode, all editors are animated with real time values of the edited objects:



- In the Program Variables Window at Run time, the value of a variable is displayed in text format in the "Value" column.

### Forcing a variable:

At run time, double click on a variable name or press ENTER key when it is selected in the Program Variables Window or in the Logic Editor Area to open the Variable Debug Window.



The Variable Debug Window has the option to Force, Lock, or Unlock the value of variable as entered in the variable field. The window also gives the value of the variable in binary format.

In the Logic Editing Area for each of the languages, the debug is displayed in following forms:

- Values of variables, contacts and coils are displayed in FBD and LD diagrams. Double click on a variable name to force or lock the variable.
- Step activities (tokens) are displayed in the SFC editor
- In the text (ST or IL) editor, place the mouse cursor on a variable name to display its real time value in a tooltip. Double click on the variable name with the SHIFT key pressed to force or lock the variable.

### **Using Data Watch window - Variable Lists**

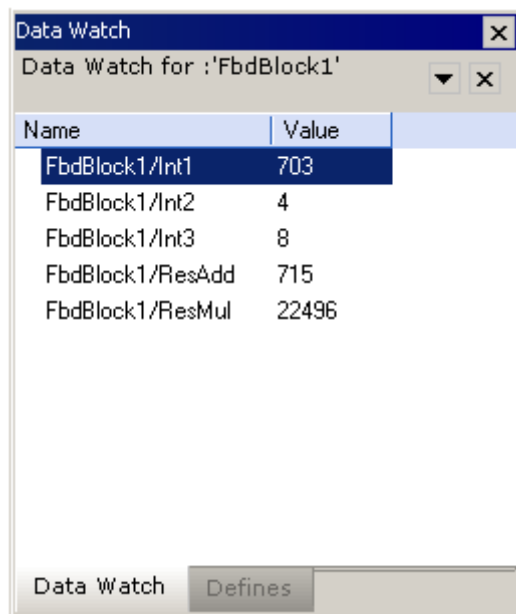
The Data Watch window is available in the Debug mode only.



Press this button in the Main Window to open the Data Watch window.

It can also be accessed from Tools | Data Watch Window.

It is a powerful monitoring tool that enables you to spy variables of the application at Run time. The watch window enables you to build list of variables to be monitored that get saved along with the program.



**Insert Variable:** Right click in the Data Watch Window to insert a variable and select Insert Variable or use keyboard shortcut Ins. Alternatively, you can drag variables from program editors or from the Program Variables Window to the Data Watch window to insert them in the list.

**Move variables within the list:** Use the "Move Up/ Move Down" commands to change the order of variables in the list.

**Hexadecimal Display:** You can right click and select Hexadecimal Display to display the variables in hexadecimal format. This feature is very useful when you spy strings with a large length or including non printable characters.

## Tools

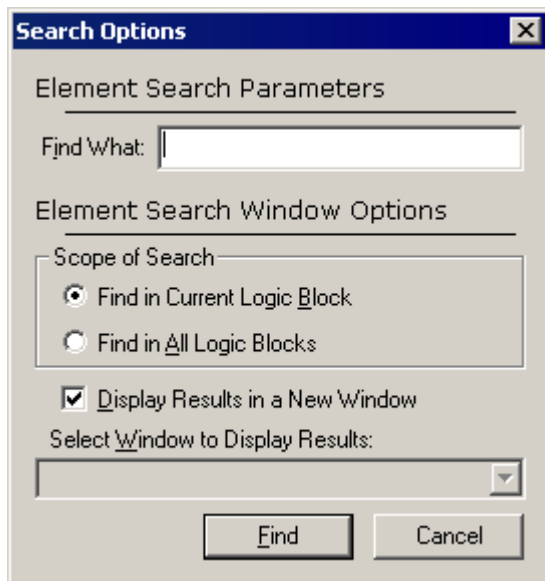
### ***Finding Elements in IEC modules***

For IEC Editors, there are two find options available, Edit | Find and Edit | Find in Logic Modules which are mentioned below:

The Find option searches the text in the current logic block and stops when the next occurrence of the same is found. The user needs to select the text and press enter to find it again. This option also wraps searches in the current logic block.



The Find in Logic Modules has the option of searching the text in either current logic module or all the logic modules. The Search results are displayed in upto two separate docking windows. The user can also select the window in which search results search results should be shown. Double clicking a reported location in the search window goes to the element without closing the search window.



Search results are displayed in upto two separate docking windows. The user can also select the window in which search results search results should be shown.

Double clicking a reported location in the search window goes to the element without closing the search window.

Search 1

Program Name: "untitled1", Search Type:Current Logic Block

Search Results For: "add"

Logic Block Name	Location
------------------	----------

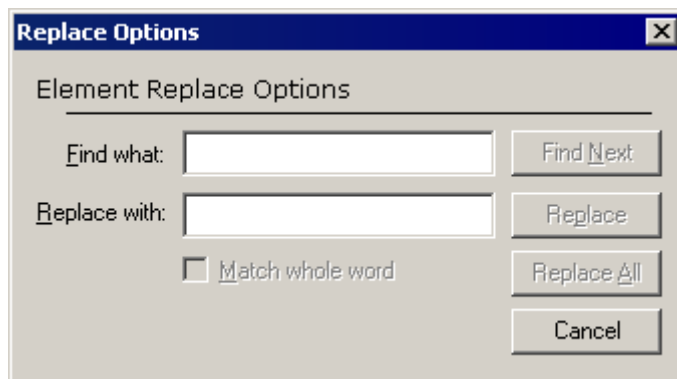
Search 1

### **How to search and Replace in IEC Modules**

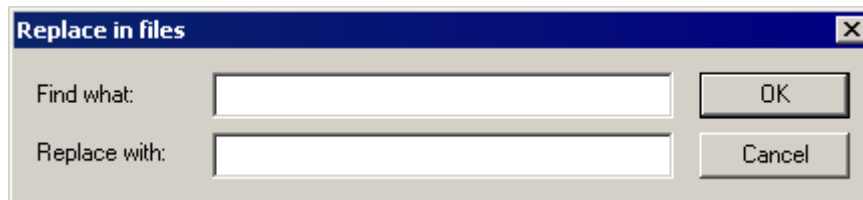
This feature allows the user to locate and change all occurrences of a particular variable in all logic modules of an IEC program. For example, all occurrences of Var1 can be changed to MyVar.

For IEC Editors, two replace options are available, Edit | Replace and Edit | Replace in Logic Modules which are mentioned below:

The Replace option searches the text in the current logic block and stops when the next occurrence of the same is found. The user needs to select the text and press enter to find it again. This option also wraps searches in the current logic block.



The Replace in Logic Modules has the option of replacing all the occurrences of the text in either current logic module or selected logic modules. The occurrences replaced are displayed in output window. Double clicking a reported location in the search window goes to the element without closing the search window.



#### **Find What**

In the Find What area enter the variable name to be replaced.

#### **Replace With**

In the Replace With area, enter the variable name to be used as the replacement.

#### **Match whole word**

When this checkbox is checked, full word would be matched. For example, if there are three variables NewVar, NewVar1 and NewVar2 defined, Find What string is NewVar and Replaced With string is MyVar. When this checkbox is unchecked, all occurrences of NewVar, NewVar1 and NewVar2 will be replaced with MyVar. If this option is checked, only NewVar occurrences will be replaced with MyVar.

Note: text replaced by these options can be reverted back.

## Languages

### Programming Languages - Reference guide

Refer to the following pages for an overview of the IEC61131-3 programming languages:

Program organization units

Data types

Structures

Variables

Arrays

Constant expressions

Conditional compiling

SFC: Sequential Function Chart

FBD: Function Block Diagram

LD: Ladder Diagram

ST: Structured Text

IL: Instruction List

The following topics detail the set of programming features and standard blocks:

Basic operations



## **Programming Languages**

### ***Programming languages - Overview***

Below are the available programming languages of the IEC61131-3 standard:

SFC: Sequential Function Chart

FBD: Function Block Diagram

LD: Ladder Diagram

ST: Structured Text

IL: Instruction List

You have to select a language for each program or User Defined Function Block of the application.

## **SFC**

### **Sequential Function Chart (SFC)**

The SFC language is a state diagram. Graphical steps are used to represent stable states, and transitions describe the conditions and events that lead to a change of state. Using SFC highly simplifies the programming of sequential operations as it saves a lot of variables and tests just for maintaining the program context.

#### **Important Note**

You should not use SFC as a decision diagram. Using a step as a point of decision and transitions as conditions in an algorithm should never appear in a SFC chart. Using SFC as a decision language leads to poor performance and complicated charts. ST should be preferred when programming a decision algorithm that has no sense in terms of "program state".

Below are basic components of an SFC chart:

#### *Chart:*

- Steps and initial steps
- Transitions and divergences
- Parallel branches
- Macro-steps
- Jump to a step

#### *Programming:*

- Actions within a step
- Programming a transition condition
- How SFC is executed

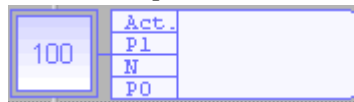
### SFC Steps

A step represents a stable state. It is drawn as a square box in the SFC chart. Each must step of a program is identified by a unique number. At run time, a step can be either active or inactive according to the state of the program.

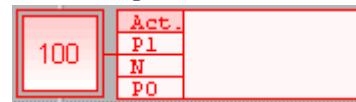
**Note:** To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.

All actions linked to the steps are executed according to the activity of the step.

#### Inactive step



#### Active step



In conditions and actions of the SFC program, you can test the step activity by specifying its name ("GS" plus the step number) followed by ".X". For example:

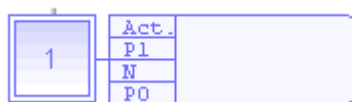
GS100.X is TRUE if step 100 is active  
(expression has the BOOL data type)

You can also test the activity time of a step, by specifying the step name followed by ".T". It is the time elapsed since the activation of the step. When the step is de-activated, this time remains unchanged. It will be reset to 0 on the next step activation. For example:

GS100.T is the time elapsed since step 100 was activated  
(expression has the TIME data type)

### Initial Steps

Initial steps represent the initial situation of the chart when the program is started. There must be at least one initial step in each SFC chart. An initial step is marked with a double line as shown below:

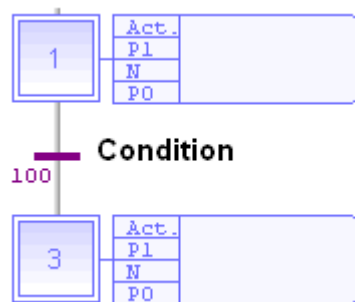


## SFC Transitions

Transitions represent a condition that changes the program activity from a step to another.

**Note:** To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.

The transition is marked by a small horizontal line that crosses a link drawn between the two steps:



Each transition is identified by a unique number in the SFC program.

Each transition must be completed with a boolean condition that indicates if the transition can be crossed. The condition is a BOOL expression.

In order to simplify the chart and reduce the number of drawn links, you can specify the activity flag of a step (GSnnn.X) in the condition of the transition.

Transitions define the dynamic behaviour of the SFC chart, according to the following rules:

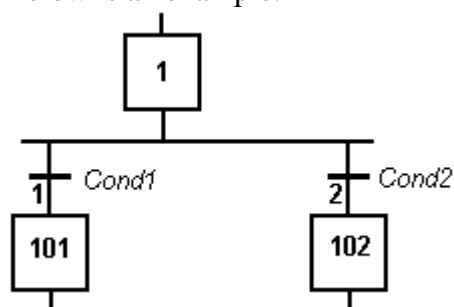
- A transition is crossed if:
  - its condition is TRUE
  - and if all steps linked to the top of the transition (before) are active
- When a transition is crossed:
  - all steps linked to the top of the transition (before) are de-activated
  - all steps linked to the bottom of the transition (after) are activated

## Divergences

It is possible to link a step to several transitions and thus create a divergence. The divergence is represented by a horizontal line. Transitions after the divergence represent several possible changes in the situation of the program.

All conditions are considered as exclusive, according to a "left to right" priority order. It means that a transition is considered as FALSE if at least one of the transitions connected to the same divergence on its left side is TRUE.

Below is an example:



Transition 1 is crossed if:

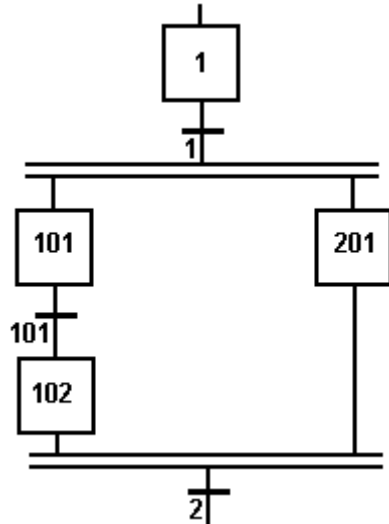
step 1 is active  
and Cond1 is TRUE

Transition 2 is crossed if:

step 1 is active  
and Cond2 is TRUE  
and Cond1 is FALSE

### SFC Parallel Branches

Parallel branches are used in SFC charts to represent parallel operations. Parallel branches occur when more than several steps are connected after the same transition. Parallel branches are drawn as double horizontal lines:



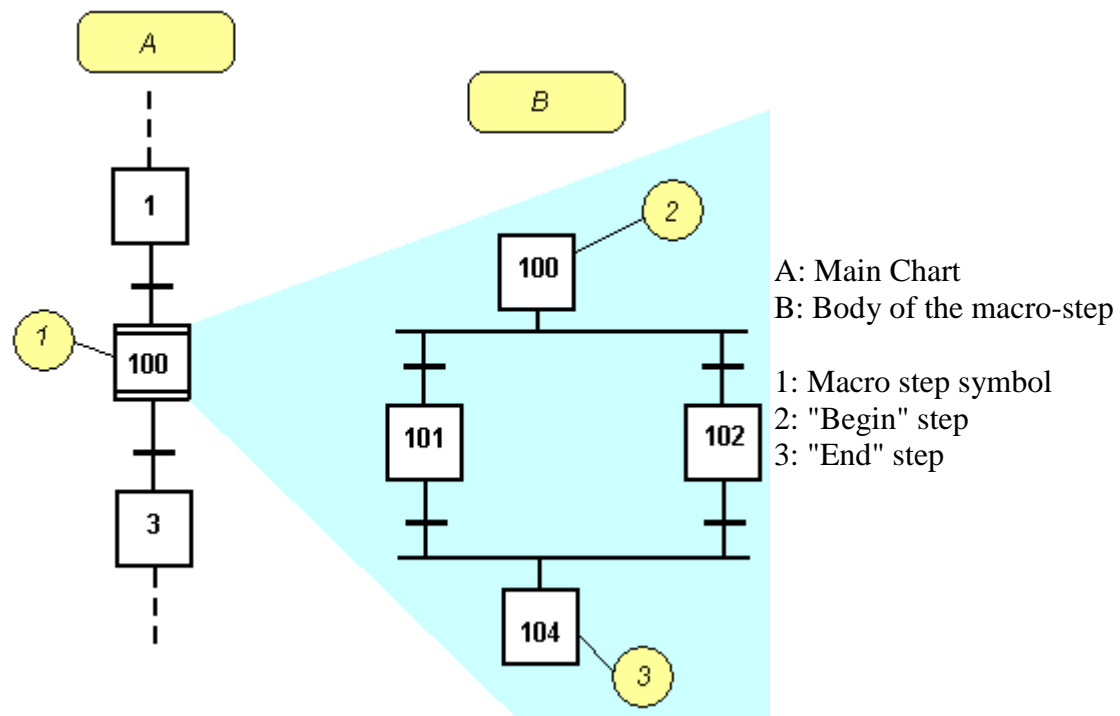
When the transition before the divergence (1 on this example) is crossed, all steps beginning the parallel branches (101 and 201 here) are activated. Sequencing of parallel branches may take different timing according to each branch execution. The transition after the convergence (2 on this example) is crossed when all the steps connected before the convergence line (last step of each branch) are active. The transition indicates a synchronization of all parallel branches. If needed, a branch may be finished with an "empty" step (with no action). It represents the state where the branch "waits" for the other ones to be completed.

You must take care of the following rules when drawing parallel lines in order to avoid dead locks in the execution of the program:

- All branches must be connected to the divergence and the convergence.
- An element of a branch must not be connected to an element outside the divergence.

### SFC Macro Steps

A macro step is a special symbol that represents, within a SFC chart, a part of the chart that begins with a step and ends with a step. The body of the macro-step must be declared in the same program. The body of a macro-step begins with a special "begin" step with no link before, and ends with a special "end" step with no link after. The symbol of the macros step in the main chart has double horizontal lines:



Important notes:

- The macro-step symbol and the beginning step must have the same number.
- The body of the macro-step should have no link with other parts of the main diagram (must be connex).
- A macro step is not a "sub program". It is just a drawing features that enables you to make clearer charts. You should never insert several macro-step symbols referring to the same macro-step body.

### **Jump to a SFC step**

"Jump" symbols can be used in SFC charts to represent a link from a transition to a step without actually drawing it. The jump is represented by an arrow identified with the number of the target step.

Note: To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.

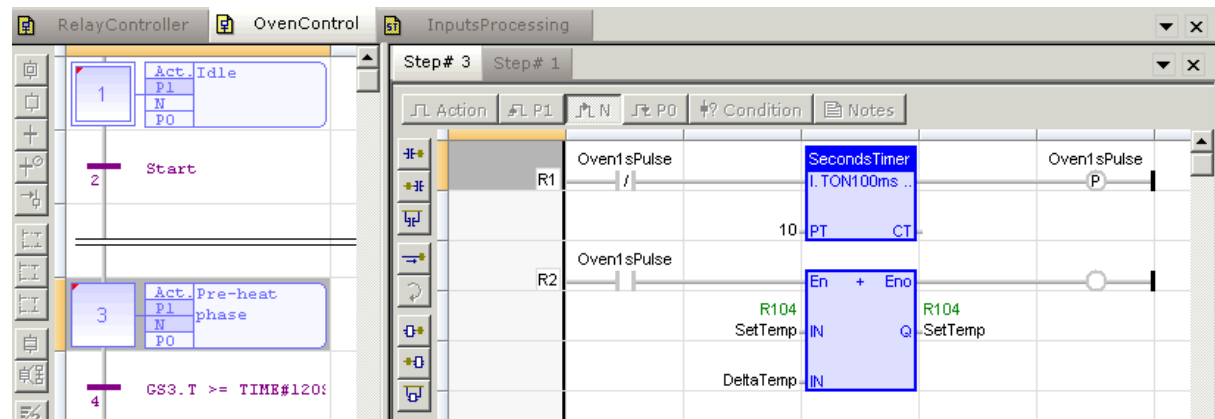
You cannot insert a jump to a transition as it may lead to a non explicit convergence of parallel branches (several steps leading to the same transition) and generally leads to mistakes due to a bad understanding of the chart.

## Actions in an SFC Step

Each step has a list of action blocks, that are instructions to be executed according to the activity of the step. Actions can be simple boolean or SFC actions, that consists in assigning a boolean variable or control a child SFC program using the step activity, or action blocks entered using another language (FBD, LD, ST or IL).

Example of an action in ST Language:

Example of an action in LD Language:



Simple boolean actions:

Below are the possible syntaxes you can use within an SFC step to perform a simple boolean action:

BoolVar (N);	Forces the variable "BoolVar" to TRUE when the step is activated, and to FALSE when the step is de-activated.
BoolVar (S);	Sets the variable "BoolVar" to TRUE when step is activated
BoolVar (R);	Sets the variable "BoolVar" to FALSE when step is activated
/ BoolVar;	Forces the variable "BoolVar" to FALSE when the step is activated, and to TRUE when the step is de-activated.

Programmed action blocks:

Programs in other languages (FBD, LD, ST or IL) can be entered to describe an SFC step action. There are three main types of programmed action blocks, that correspond to the following identifiers:

P1	Executed only once when the step becomes active
N	Executed on each cycle while the step is active
P0	Executed only once when the step becomes inactive

**i3 Configurator** provides you templates for entering P1, N and P0 action blocks in either ST, LD or FBD language. Alternatively, you can insert action blocks



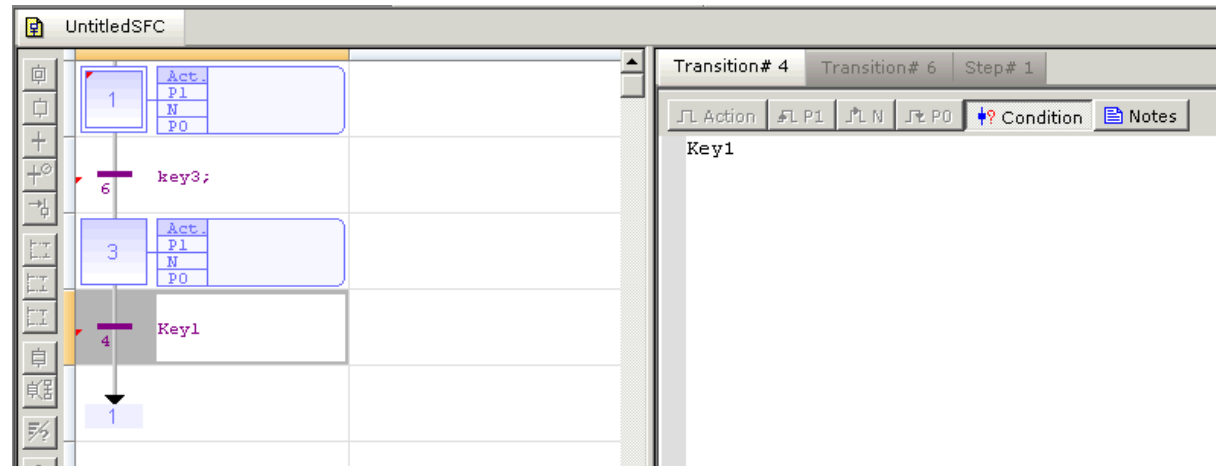
programmed in ST language directly in the list of simple actions, using the following syntax:

```
ACTION ( qualifier ) :  
    statements...  
END_ACTION;
```



Where *qualifier* is "P1", "N" or "P0".

### Condition of a SFC Transition

Each SFC transitions must have a boolean condition that indicates if the transition can be crossed. The condition is a boolean expression that can be programmed either in ST or LD language.



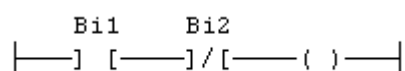
The following conditions can be attributed to a Transition

-  Transition[T]: Insert a transition
-  Set timer on transition: Set timer to a transition

A Timed Transition will execute the next Step after the time has elapsed. In ST language, enter a boolean expression. It can be a complex expression including function calls and parenthesis. For example:

bForce AND (bAlarm OR min (iLevel, 1) <> 1)

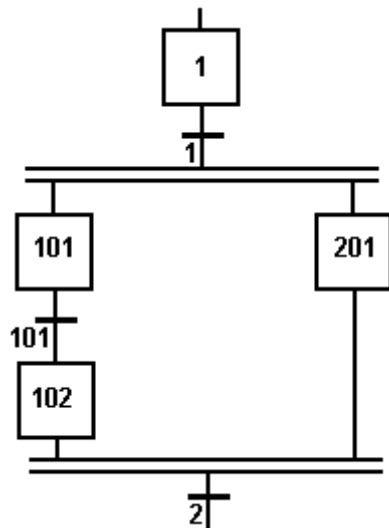
In LD language, the condition is represented by a single rung. The coil at the end of the rung represents the transition and should have no symbol attached. For example:



### Execution at Runtime

SFC programs are executed sequentially within a target cycle, according to the order defined when entering programs in the hierarchy tree.

Within a chart, all valid transitions are evaluated first, and then actions of active steps are performed. The chart is evaluated from the left to the right and from the top to the bottom. Below is an example:



Execution order:

- Evaluate transitions:
- 1, 101, 2
- Manage steps:
- 1, 101, 201, 102

In case of a divergence, all conditions are considered as exclusive, according to a "left to right" priority order. It means that a transition is considered as FALSE if at least one of the transitions connected to the same divergence on its left side is TRUE.

The initial steps define the initial status of the program when it is started. All top level (main) programs are started when the application starts.

The evaluation of transitions leads to changes of active steps, according to the following rules:

- A transition is crossed if:
  - its condition is TRUE
  - and if all steps linked to the top of the transition (before) are active
- When a transition is crossed:
  - all steps linked to the top of the transition (before) are de-activated
  - all steps linked to the bottom of the transition (after) are activated

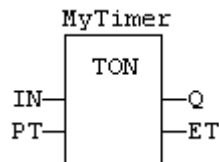
Important note:

Execution of SFC within the target is sampled according to the target cycles. When a transition is crossed within a cycle, the following steps are activated, and the evaluation of the chart will continue on the next cycle. If several consecutive transitions are TRUE within a branch, only one of them is crossed within one target cycle.

### **Function Block Diagram (FBD)**

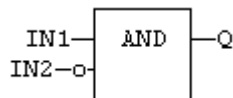
A Function Block Diagram is a data flow between constant expressions or variables and operations represented by rectangular blocks. Operations can be basic operations, function calls, or function block calls.

The name of the operation or function, or the type of function block is written within the block rectangle. In case of a function block call, the name of the called instance must be written upon the block rectangle, such as in the example below:



The data flow may represent values of any data type. All connections must be from input and outputs points having the same data type. In case of a boolean connection, you can use a connection link terminated by a small circle, that indicates a boolean negation of the data flow. Below is an example:

(\* use of a negated link: Q is IN1 AND NOT IN2 \*)



The data flow must be understood from left to the right and from the top to the bottom. It is possible to use labels and jumps to change the default data flow execution.

### **LD symbols**

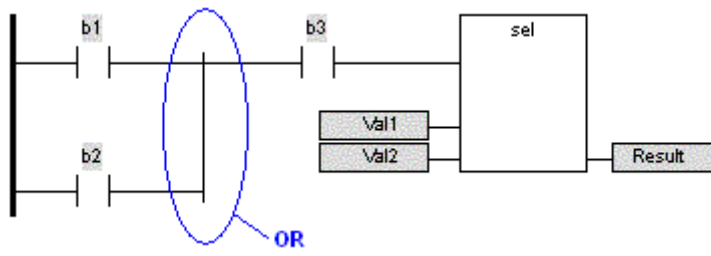
LD symbols may also be entered in FBD diagrams and linked to FBD objects. Refer to the following sections for further information about components of the LD language:

Contacts

Coils

Power Rails

Special vertical lines are available in FBD language for representing the merging of LD parallel lines. Such vertical lines represent a OR operation between the connected inputs. Below is an example of an OR vertical line used in a FBD diagram:



## LD

### Ladder Diagram (LD)

A Ladder Diagram is a list of *rungs*. Each rung represents a boolean data flow from a power rail on the left to a power rail on the right. The left power rail represents the TRUE state. The data flow must be understood from the left to the right. Each symbol connected to the rung either changes the rung state or performs an operation. Below are possible graphic items to be entered in LD diagrams:

Power Rails

Contacts and Coils

Operations, Functions and Function blocks, represented by rectangular blocks

Labels and Jumps

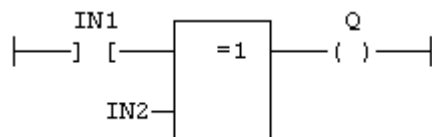
Use of the "EN" input and the "ENO" output for blocks

The rung state in a LD diagram is always boolean. Blocks are connected to the rung with their first input and output. This implies that special "EN" and "ENO" input and output are added to the block if its first input or output is not boolean.

The "EN" input is a condition. It means that the operation represented by the block is not performed if the rung state (EN) is FALSE. The "ENO" output always represents the same status as the "EN" input: the rung state is not modified by a block having an ENO output.

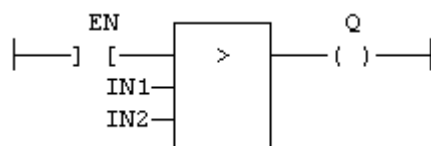
Below is the example of the "XOR" block, having boolean inputs and outputs, and requiring no EN or ENO pin:

(\* First input is the rung. The rung is the output \*)



Below is the example of the ">" (greater than) block, having non boolean inputs and a boolean output. This block has an "EN" input in LD language:

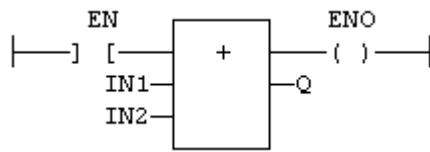
(\* The comparison is executed only if EN is TRUE \*)



Finally, below is the example of an addition, having only numerical arguments. This block has both "EN" and "ENO" pins in LD language:

(\* The addition is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



## Contacts

### Coils | Power Rails

Contacts are basic graphic elements of the LD language. A contact is associated to a boolean variable written upon its graphic symbol.

A contact sets the state of the rung on its right side, according to the value of the associated variable to itself and the rung state on its left side i.e. If power flows from the left side and the contact value is True, then power will flow to the right side of the contact.

Below are the possible contact symbols and how they change the rung state:

<code>BoolVariable</code> —] [—	Normal: the rung state on the right is the boolean AND between the rung state on the left and the associated variable.
<code>BoolVariable</code> —] / [—	Negated: the rung state on the right is the boolean AND between the rung state on the left and the negation of the associated variable.
<code>BoolVariable</code> —] P [—	Positive pulse: the rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from FALSE to TRUE (rising edge).
<code>BoolVariable</code> —] N [—	Negative pulse: the rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from TRUE to FALSE (falling edge).

Two serial normal contacts represent an AND operation.

Two contacts in parallel represent an OR operation.



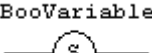
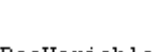
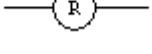
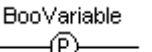


## Coils

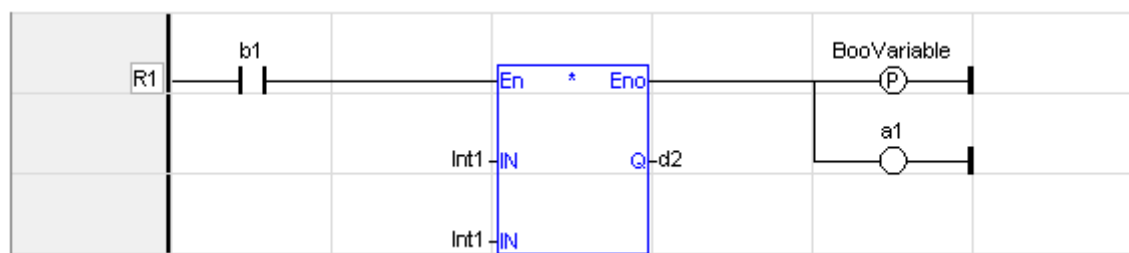
Contacts | Power Rails

Coils are basic graphic elements of the LD language. A coil is associated to a boolean variable written upon its graphic symbol. A coil performs a change of the associated variable according to the rung state on its left side i.e. A coil is energized or de-energized based upon the power flow to it from the rung to which it is associated.

Below are the possible coil symbols and how they change the rung state:

<div> <div>BoolVariable</div>  </div>	Normally Open: The associated variable is forced to the value of the rung state on the left of the coil.
<div> <div>BoolVariable</div>  </div>	Normally Closed: The associated variable is forced to the negation of the rung state on the left of the coil.
<div> <div>BoolVariable</div>  </div>	Set: The associated variable is forced to TRUE if the rung state on the left goes TRUE. It will remain forced to TRUE even if the power flow through the associated rung becomes inactive.(No action if the rung state is FALSE).
<div> <div>BoolVariable</div>  </div>	Reset: The associated variable is forced to FALSE if the rung state on the left goes TRUE. It will remain forced to FALSE even if the power flow through the associated rung becomes inactive (No action if the rung state is FALSE)
<div> <div>BoolVariable</div>  </div>	Positive Triggered: The associated variable stays TRUE for one cycle after the rising-edge is detected for the associated rung. It will go back to FALSE in the next execution cycle.
<div> <div>BoolVariable</div>  </div>	Negative Triggered: The associated variable stays TRUE for one cycle after the falling-edge is detected for the associated rung. It will go back to FALSE in the next execution cycle.

Coils may be connected in parallel if a single rung is supposed to activate more than one coil.



## Power Rails

Contacts | Coils

Vertical power rails are used in LD language for designing the limits of a rung.

The power rail on the left represents the TRUE value and initiates the rung state. The power rail on the right receives connections from the coils and has no influence on the execution of the program.

Power rails can also be used in FBD language. Only boolean objects can be connected to left and right power rails. Power rails can be resized before elements are connected to them.

### **Structured Text (ST)**

ST is a structured literal programming language. A ST program is a list of *statements*. Each statement describes an action and must end with a semi-colon (";").

The presentation of the text has no meaning for a ST program. You can insert blank characters and line breaks where you want in the program text.

### **Comments**

Comment texts can be entered anywhere in a ST program. Comment texts have no meaning for the execution of the program. A comment text must begin with "(" and end with ")". Comments can be entered on several lines (i.e. a comment text may include line breaks). Comment texts cannot be nested.

### **Expressions**

Each statement describes an action and may include evaluation of complex expressions. An expression is evaluated:

- from the left to the right
- according to the default priority order of operators
- the default priority can be changed using parenthesis

Arguments of an expression can be:

- declared variables
- constant expressions
- function calls

### **Statements**

Below are available basic statements that can be entered in a ST program:

- assignment
- function block calling

Below are the available conditional statements in ST language:

- IF / THEN / ELSE (simple binary switch)
- CASE (enumerated switch)

Below are the available statements for describing loops in ST language:

- WHILE (with test on loop entry)
- REPEAT (with test on loop exit)
- FOR (enumeration)

### Instruction List (IL)

A program written in IL language is a list of *instructions*. Each instruction is written on one line of text. An instruction may have one or more *operands*. Operands are variables or constant expressions. Each instruction may begin with a label, followed by the ":" character. Labels are used as destination for jump instructions.

**i3 Configurator** allows you to mix ST and IL languages in textual program. ST is the default language. When you enter IL instructions, the program must be entered between "BEGIN\_IL" and "END\_IL" keywords, such as in the following example

```
BEGIN_IL
  LD  var1
  ST  var2
END_IL
```

### Comments

Comment texts can be entered at the end of a line containing an instruction. Comment texts have no meaning for the execution of the program. A comment text must begin with "(" and end with "\*". Comments may also be entered on empty lines (with no instruction), and on several lines (i.e. a comment text may include line breaks). Comment texts cannot be nested.

### Data flow

An IL complete statement is made of instructions for:  
first: evaluating an expression (called *current result*) then: use the current result for performing actions

### Evaluation of expressions

The order of instructions in the program is the one used for evaluating expressions, unless parenthesis are inserted. Below are the available instructions for evaluation of expressions:

<i>instruction</i>	<i>operand</i>	<i>meaning</i>
LD / LDN	any type	loads the operand in the current result
AND (&)	boolean	AND between the operand and the current result
OR / ORN	boolean	OR between the operand and the current result
XOR / XORN	boolean	XOR between the operand and the current result
ADD	numerical	adds the operand and the current result
SUB	numerical	subtract the operand from the current result
MUL	numerical	multiply the operand and the current result
DIV	numerical	divides the current result by the operand
GT	numerical	compares the current result with the operand
GE	numerical	compares the current result with the operand
LT	numerical	compares the current result with the operand
LE	numerical	compares the current result with the operand

EQ	numerical	compares the current result with the operand
NE	numerical	compares the current result with the operand
Function call	func. arguments	calls a function
Parenthesis		changes the execution order

*Notes:* Instructions suffixed by "N" uses the boolean negation of the operand.

### **Actions**

The following instructions perform actions according to the value of current result.  
Some of these instructions do not need a current result to be evaluated:

<i>instruction</i>	<i>operand</i>	<i>meaning</i>
ST / STN	any type	stores the current result in the operand
JMP	label	jump to a label - no current result needed
JMPC / JMPCN	label	jump to a label if the current result is TRUE
S	boolean	sets the operand to TRUE if the current result is TRUE
R	boolean	sets the operand to FALSE if the current result is TRUE
CAL	f. block	calls a function block (no current result needed)
CALC / CALCN	numerical	calls a function block if the current result is TRUE

*Notes:* Instructions suffixed by "N" uses the boolean negation of the operand.

## Program Organization Units

An application is a list of programs. Programs are executed sequentially within the target cycle, according to the following model:

```
Begin cycle
| exchange I/Os
| execute first program
| ...
| execute last program
| wait for cycle time to be elapsed
End Cycle
```

Programs are executed according to the order defined by the user. All SFC programs must be grouped (it is not possible to insert a program in FBD, LD, ST or IL in between two SFC programs). The number of programs in an application is limited to 255. Each program is entered using a language chosen when the program is created. Possible languages are Sequential Function Chart (SFC), Function Block Diagram (FBD), Ladder Diagram (LD), Structured Text (ST) or Instruction List (IL).

Programs must have unique names. The name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A program should not have the same name as a declared variable. The name of a program should begin by a letter or an underscore (" \_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a name. Naming is case insensitive. Two names with different cases are considered as the same.

### ***Sub-programs***

The list of programs may be completed by "Sub-programs". Sub-programs are described using FBD, LD, ST or IL language, and can be called by the programs of the application. Input and output parameters plus local variables of a sub-program are declared in the Program Variables Window as local variables of the sub-program.

A sub-program may call another sub-program.

Local variables of a sub program are not instantiated. This means that the sub-programs always work on the same set of local variables. Local variables of a sub-program keep their value among various calls. The code of a sub-program is not duplicated when called several times by parent programs.

A sub-program cannot have more than 32 input parameters or 32 output parameters. A parameter of a sub-program cannot be an instance of a function block.

## Data Types

Below are the available basic data types:

<b>BOOL</b>	Boolean (bit) - can be FALSE or TRUE - stored on 1 byte
<b>SINT</b>	Small signed integer on 8 bits (from -128 to +127)
<b>USINT</b>	Small unsigned integer on 8 bits (from 0 to +255)
<b>BYTE</b>	<i>Same as USINT</i>
<b>INT</b>	Signed integer on 16 bits (from -32768 to +32767)
<b>UINT</b>	Unsigned integer on 16 bits (from 0 to +65535)
<b>WORD</b>	<i>Same as UINT</i>
<b>DINT</b>	Signed integer on 32 bits (from -2147483648 to +2147483647)
<b>UDINT</b>	Unsigned integer on 32 bits (from 0 to +4294967295)
<b>DWORD</b>	<i>Same as UDINT</i>
<b>REAL</b>	Single precision floating point - stored on 32 bits
<b>TIME</b>	Time of day - less than 24h - accuracy is 1ms
<b>STRING</b>	Variable length string with declared maximum length The declared maximum length cannot exceed 255 characters

## Variables

All variables used in programs must be first declared in the Program Variables Window. Each variable belongs to a group and is must be identified by a unique name within its group.

### Groups

A group is a set of variables. A group either refers to a physical class of variables, or identifies the variables local to a program or user defined function block. Below are the possible groups:

<b>GLOBAL</b>	Internal variables known by all programs
<b>RETAIN</b>	Non volatile internal variables known by all programs
<b>PROGRAM<sub>xxx</sub></b>	All internal variables local to a program (the name of the group is the name of the program)

### Data type and dimension

Each variable must have a valid data type. It can be either a basic data type or a function block. In that case the variable is an instance of the function block. Instances of function blocks can refer either to a standard or "C" embedded block.

If the selected data type is STRING, you must specify a maximum length, that cannot exceed 255 characters.

Refer to the list of available data types for more informartion. Refer to the section describing function blocks for further information about how to use a function instance.

Additionally, you can specify dimension(s) for an internal variable, in order to declare an array. Arrays have at most 3 dimensions. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by *ArrayName*[0]. You cannot declare arrays of function block instances. The total number of items in an array (merging all dimensions) cannot exceed 65535.

### Naming a variable

A variable must be identified by a unique name within its parent group. The variable name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A variable should not have the same name as a program or a user defined function block.

The name of a variable should begin by a letter or an underscore ("\_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a variable name. Naming is case insensitive. Two names with different cases are considered as the same.



### ***Attributes of a variable***

For each internal variable, you can select the "**Read Only**".

## Arrays

You can specify dimension(s) for internal variables, in order to declare arrays. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by *ArrayName*[0].

To declare an array, enter its dimension in the corresponding column of the Program Variables Window. For a multi-dimension array, enter dimensions separated by comas (ex: **2,10,4**).

### ***Use in ST and IL languages:***

To specify an item of an array in ST and IL language, enter the name of the array followed by the index(es) entered between "[" and "]" characters. For multidimension arrays, enter indexes separated by comas. Indexes may be either constant or complex expressions. Below are some examples in ST language:

```
TheArray[1,7] := value;  
result := SingleArray[i + 2];
```

### ***Use in FBD and LD languages:***

In graphical languages, the following blocks are available for managing array elements:

[I]>>	get value of an item in a single dimension array
[I, J]>>	get value of an item in a two dimension array
[I, J, K]>>	get value of an item in a three dimension array
>>[I]	set value of an item in a single dimension array
>>[I, J]	set value of an item in a two dimension array
>>[I, J, K]	set value of an item in a three dimension array

For "get" blocks, the first input is the array and the output is the value of the item. Other inputs are indexes in the array.

For "put" blocks, the first input is the forced value and the second input is the array. Other inputs are indexes in the array.

### ***Restrictions:***

- Arrays have at most 3 dimensions.
- All indexes are 0 based.
- You cannot declare arrays of function block instances or structures.
- The total number of items in an array (merging all dimensions) cannot exceed 65535.
- Array elements cannot be specified on a contact or a coil

## Constant Expressions

Constant expressions can be used in all languages for assigning a variable with a value. All constant expressions have a well defined data type according to their semantics. If you program an operation between variables and constant expressions having inconsistent data types, it will lead to syntactic errors when the program is compiled. Below are the syntactic rules for constant expressions according to possible data types:

### ***BOOL: Boolean***

There are only two possible boolean constant expressions. They are reserved keywords **TRUE** and **FALSE**.

### ***SINT: Small (8 bit) Integer***

Small integer constant expressions are valid integer values(between -128 and 127) and must be prefixed with "**SINT#**". All integer expressions having no prefix are considered as DINT integers.

### ***USINT / BYTE: Unsigned 8 bit Integer***

Unsigned small integer constant expressions are valid integer values(between 0 and 255) and must be prefixed with "**USINT#**". All integer expressions having no prefix are considered as DINT integers.

### ***INT: 16 bit integer***

16 bit integer constant expressions are valid integer values(between -32768 and 32767) and must be prefixed with "**INT#**". All integer expressions having no prefix are considered as DINT integers.

### ***UINT / WORD: Unsigned 16 bit integer***

Unsigned 16 bit integer constant expressions are valid integer values(between 0 and 255) and must be prefixed with "**UINT#**". All integer expressions having no prefix are considered as DINT integers.

### ***DINT: 32 bit (default) integer***

32 bit integer constant expressions must be valid numbers between -2147483648 to +2147483647. DINT is the default size for integers: such constant expressions do not need any prefix. You can use "**2#**", "**8#**" or "**16#**" prefixes for specifying a number in respectively binary, octal or hexadecimal basis.

### ***UDINT / DWORD: Unsigned 32 bit integer***

Unsigned 32 bit integer constant expressions are valid integer values(between 0 and 4294967295) and must be prefixed with "UDINT#". All integer expressions having no prefix are considered as DINT integers.

### ***REAL: Single precision floating point value***

Real constant expressions must be valid number, and must include a dot ("."). If you need to enter a real expression having an integer value, add ".0" at the end of the number. You can use "F" or "E" separators for specifying the exponent in case of a scientist representation. REAL is the default precision for floating points: such expressions do not need any prefix.

### ***TIME: Time of day***

Time constant expressions represent durations that must be less than 24 hours. Expressions must be prefixed by either "TIME#" or "T#". They are expressed as a number of hours followed by "h", a number of minutes followed by "m", a number of seconds followed by "s", and a number of milliseconds followed by "ms". The order of units (hour, minutes, seconds, milliseconds) must be respected. You cannot insert blank characters in the time expression. There must be at least one valid unit letter in the expression.

### ***STRING: Character string***

String expressions must be written between single quote marks. The length of the string cannot exceed 255 characters. You can use the following sequences to represent a special or not printable character within a string:

<b>\$\$</b>	a "\$" character
<b>\$'</b>	a single quote
<b>\$T</b>	a tab stop (ASCII code 9)
<b>\$R</b>	a carriage return character (ASCII code 13)
<b>\$L</b>	a line feed character (ASCII code 10)
<b>\$N</b>	carriage return plus line feed characters (ASCII codes 13 and 10)
<b>\$P</b>	a page break character (ASCII code 12)
<b>\$xx</b>	any character (xx is the ASCII code expressed on two hexadecimal digits)

### ***Examples***

Below are some examples of valid constant expressions

<b>TRUE</b>	TRUE boolean expression
<b>FALSE</b>	FALSE boolean expression
<b>SINT#127</b>	small integer
<b>INT#2000</b>	16 bit integer
<b>123456</b>	DINT (32 bit) integer
<b>16#abcd</b>	DINT integer in hexadecimal basis
<b>0.0</b>	0 expressed as a REAL number
<b>1.002E3</b>	1002 expressed as a REAL number in scientist format

<b>T#23h59m59s999ms</b>	maximum TIME value
<b>TIME#0s</b>	null TIME value
<b>T#1h123ms</b>	TIME value with some units missing
<b>'hello'</b>	character string
<b>'name\$Tage'</b>	character string with two words separated by a tab
<b>'I\$m here'</b>	character string with a quote inside (I'm here)
<b>'x\$00y'</b>	character string with two characters separated by a null character (ASCII code 0)

Below are some examples of typical errors in constant expressions

<b>BooVar := 1;</b>	0 and 1 cannot be used for booleans
<b>1a2b</b>	basis prefix ("16#") omitted
<b>T#12</b>	Time unit missing
<b>'I'm here'</b>	quote within a string with "\$" mark omitted
<b>hello</b>	quotes omitted around a character string

## Conditional Compiling

The compiler supports conditional compiling directives in ST, IL, LD, and FBD languages. Conditional compiling directives condition the inclusion of a part of the program in the generated code. Conditional compiling is an easy way to manage several various configurations and options in a unique application programming.

Conditional compiling uses definitions as conditions. Below is the main syntax:

```
# ifdef CONDITION
    statementsYES...
#else
    statementsNO...
# endif
```

If CONDITION has been defined using #define syntax, then the "*statementsYES*" part is included in the code, else the "*statementsNO*" part is included. The "#else" statement is optional.

In ST and IL text languages, directives must be entered alone on one line line of text. In FBD language, directives must be entered as the text of network breaks. In LD language, directives must be entered on comment lines.

## Basic Operations

### *Basic Operations*

Below are the language features for basic data manipulation:

- Variable assignment
- Bit access
- Parenthesis
- Calling a function
- Calling a function block
- Calling a sub-program

Below are the language features for controlling the execution of a program:

- Labels
- Jumps
- RETURN

Below are the structured statements for controlling the execution of a program:

IF	conditional execution of statements
WHILE	repeat statements while a condition is TRUE
REPEAT	repeat statements until a condition is TRUE
FOR	execute iterations of statements
CASE	switch to one of various possible statements
EXIT	exit from a loop instruction

### ***Access to Bits of an Integer***

You can directly specify a bit within an integer variable in expressions and diagrams, using the following notation:

Variable.BitNo

#### **Where:**

Variable: is the name of an integer variable

BitNo: is the number of the bit in the integer.

#### **The variable can have one of the following data types:**

SINT, USINT, BYTE (8 bits from %x.1 to %x.8)

INT, UINT, WORD (16 bits from %x.1 to %x.16)

DINT, UDINT, DWORD (32 bits from %x.1 to %x.32)

1 always represents the least significant bit.

#### **Example:**

%R1.1, %R1000.32



## **Parenthesis ( )**

### **Assignment**

*Operator* - force the evaluation order in a complex expression.

### **Remarks**

Parenthesis are used in ST and IL language for changing the default evaluation order of various operations within a complex expression. For instance, the default evaluation of "2 \* 3 + 4" expression in ST language gives a result of 10 as "\*" operator has highest priority. Changing the expression as "2 \* ( 3 + 4 )" gives a result of 14. Parenthesis can be nested in a complex expression.

Below is the default evaluation order for ST language operations (1st is highest priority):

Unary operators:	- NOT
Multiply/Divide:	* /
Add/Subtract:	+ -
Comparisons:	< > <= >= = <>
Boolean And:	& AND
Boolean Or:	OR
Exclusive OR:	XOR

In IL language, the default order is the sequence of instructions. Each new instruction modifies the current result sequentially. In IL language, the opening parenthesis "(" is written between the instruction and its operand. The closing parenthesis ")" must be written alone as an instruction without operand.

### **ST Language**

Q := (IN1 + (IN2 / IN 3)) \* IN4;

### **FBD Language**

*Not available*

### **LD Language**

*Not available*

### **IL Language**

```
Op1: LD( IN1
      ADD( IN2
          MUL IN3
        )
      SUB IN4
```

)  
ST Q (\* Q is: (IN1 + (IN2 \* IN3) - IN4) \*)

### **Calling a Function**

A function calculates a result according to the current value of its inputs. Unlike a function block, a function has no internal data and is not linked to declared instances. A function has only one output: the result of the function. A function can be:

- a standard function (SHL, SIN...)
- a function written in "C" language and embedded on the target

### **ST Language**

To call a function block in ST, you have to enter its name, followed by the input parameters written between parenthesis and separated by comas. The function call may be inserted into any complex expression. a function call can be used as an input parameter of another function. The following example demonstrates a call to "ODD" and "SEL" functions:

(\* the following statement converts any odd integer value into the nearest even integer \*)

```
iEvenVal := SEL ( ODD( iValue ), iValue, iValue+1 );
```

### **FBD and LD Languages**

To call a function block in FBD or LD languages, you just need to insert the function in the diagram and to connect its inputs and output.

### **IL Language**

To call a function block in IL language, you must load its first input parameter before the call, and then use the function name as an instruction, followed by the other input parameters, separated by comas. The result of the function is then the current result. The following example demonstrates a call to "ODD" and "SEL" functions:

(\* the following statement converts any odd integer into "0" \*)

```
Op1: LD  iValue
      ODD
      SEL iValue, 0
      ST  iResult
```

### **Calling a Function Block   *CAL*   *CALC*   *CALNC*   *CALCN***

A function block groups an algorithm and a set of private data. It has inputs and outputs. A function block can be:

a standard function block ( RS, TON10mS...) a block written in " C" language and embedded on the target

To use a function block, you have to declare an instance of the block as a variable, identified by a unique name. Each instance of a function block has its own set of private data and can be called separately. A call to a function block instance processes the block algorithm on the private data of the instance, using the specified input parameters.

#### **ST Language**

To call a function block in ST, you have to specify the name of the instance, followed by the input parameters written between parenthesis and separated by commas. To have access to an output parameter, use the name of the instance followed by a dot '.' and the name of the wished parameter. The following example demonstrates a call to an instance of CTU function block:

(\* MyCounter is declared as an instance of CTU \*)

```
MyCounter(bCU, bReset, 200); (* calls the function block *)  
MaxCountReached:= MyCounter.Q;  
CurrentCount := MyTimer.CV;
```

#### **FBD and LD Languages**

To call a function block in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs. The name of the instance must be specified upon the rectangle of the block.

#### **IL Language**

To call a function block in IL language, you must use the CAL instruction, and use a declared instance of the function block. The instance name is the operand of the CAL instruction, followed by the input parameters written between parenthesis and separated by commas. Alternatively the CALC, CALCN or CALNC conditional instructions can be used:

```
CAL    Calls the function block  
CALC   Calls the function block if the current result is TRUE  
CALNC  Calls the function block if the current result is FALSE  
CALCN  same as CALNC
```

The following example demonstrates a call to an instance of CTU function block:

(\* MyCounter is declared as an instance of CTU \*)

Op1: CAL MyCounter (bCU, bReset, 200)

LD MyCounter.Q

ST MaxCountReached

LD MyCounter.CV

ST CurrentCount

Op2: LD bCond

CALC MyCounter (bCU, bReset, 200) (\* called only if bCond is TRUE \*)

Op3: LD bCond

CALNC MyCounter (bCU, bReset, 200) (\* called only if bCond is FALSE \*)

### **Calling a Sub-Program**

A sub-program is called by another program. Unlike function blocks, local variables of a sub-program are not instantiated, and thus you do not need to declare instances. A call to a sub-program processes the block algorithm using the specified input parameters. Output parameters can then be accessed.

#### **ST Language**

To call a sub-program in ST, you have to specify its name, followed by the input parameters written between parenthesis and separated by comas. To have access to an output parameter, use the name of the sub-program followed by a dot '.' and the name of the desired parameter:

```
MySubProg (i1, i2); (* calls the sub-program *)
```

```
Res1 := MySubProg.Q1;
```

```
Res2 := MySubProg.Q2;
```

Alternatively, if a sub-program has one and only one output parameter, it can be called as a function in ST language:

```
Res := MySubProg (i1, i2);
```

#### **FBD and LD Languages**

To call a sub-program in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs.

#### **IL Language**

To call a sub-program in IL language, you must use the CAL instruction with the name of the sub-program, followed by the input parameters written between parenthesis and separated by comas. Alternatively the CALC, CALCN or CALNC conditional instructions can be used:

CAL Calls the sub-program

CALC Calls the sub-program if the current result is TRUE

CALNC Calls the sub-program if the current result is FALSE

CALCN same as CALNC

```
Op1: CAL MySubProg (i1, i2)
```

```
LD MySubProg.Q1
```

```
ST Res1
```

```
LD MySubProg.Q2
```

```
ST Res2
```

## Labels

Jumps | RETURN

*Statement* - Destination of a Jump instruction.

### Remarks

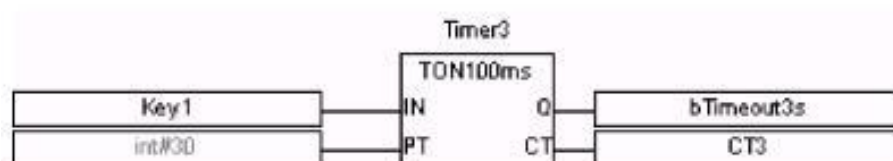
Labels are used as a destination of a jump instruction in FBD, LD or IL language. Labels and jumps cannot be used in structured ST language. A label must be represented by a unique name, followed by a colon (":"). In FBD language, labels can be inserted anywhere in the diagram, and are connected to nothing. In LD language, a label must identify a rung, and is shown on the left side of the rung. In IL language, labels are destination for JMP, JMPC, JMPCN and JMPNC instructions. They must be written before the instruction at the beginning of the line, and should index the beginning of a valid IL statement: LD (load) instruction, or unconditional instructions such as CAL, JMP or RET. The label can also be written alone on a line before the indexed instruction. In all languages, it is not mandatory that a label be a target of a jump instruction. You can also use label for marking parts of the programs in order to increase its readability.

### ST Language

*Not available*

### FBD Language

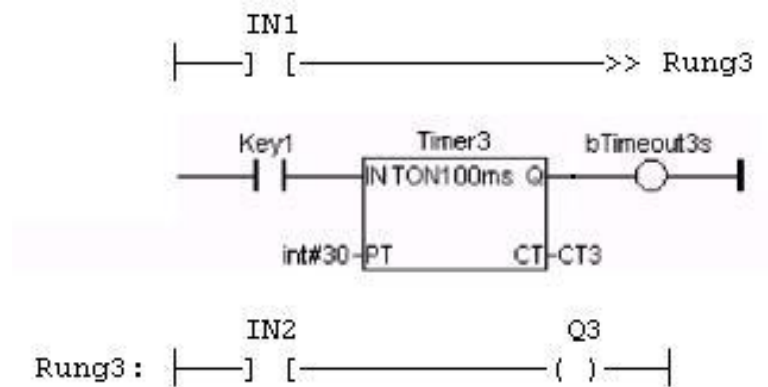
(\* In this example the TON100mS block will not be called if bEnable is TRUE \*)



TheEnd:

### LD Language

(\* In this example the second rung will be skipped if IN1 is TRUE \*)



### IL Language

```

Start: LD IN1    (* unused label - just for readability *)
      JMP TheRest (* Jump to "TheRest" if IN1 is TRUE *)

      LD IN2    (* these two instructions are not executed *)
      ST Q2     (* if IN1 is TRUE *)

TheRest: LD IN3  (* label used as the jump destination *)
      ST Q3
  
```



## Jumps JMP JMPC JMPNC JMPCN

Labels | RETURN

*Statement* - Jump to a label.

### Remarks

A jump to a label branches the execution of the program after the specified label. Labels and jumps cannot be used in structured ST language. In FBD language, a jump is represented by the ">>" symbol followed by the label name. The input of the ">>" symbol must be connected to a valid boolean signal. The jump is performed only if the input is TRUE. In LD language, the ">>" symbol, followed by the target label name, is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE. In IL language, JMP, JMPC, JMPCN and JMPNC instructions are used to specify a jump. The destination label is the operand of the jump instruction.

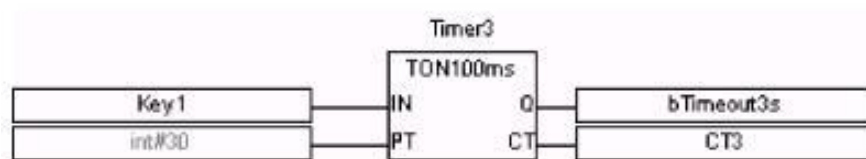
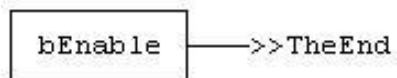
**Warning:** backward jumps may lead to infinite loops that block the target cycle.

### ST Language

*Not available*

### FBD Language

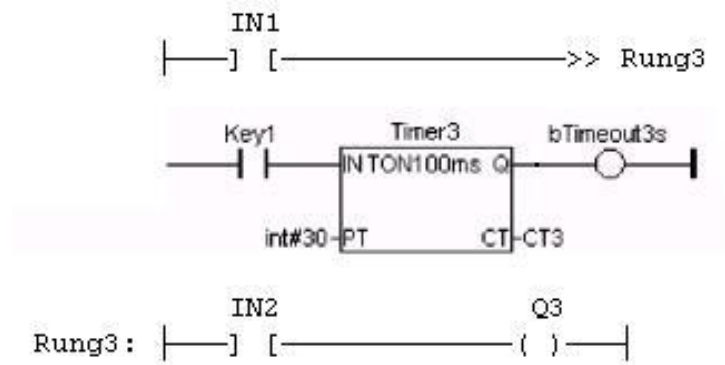
(\* In this example the TON100ms block will not be called if bEnable is TRUE \*)



TheEnd:

### LD Language

(\* In this example the second rung will be skipped if IN1 is TRUE \*)



### IL Language

Below is the meaning of possible jump instructions:

>JMP Jump always

JMPC Jump if the current result is TRUE

JMPNC Jump if the current result is FALSE

JMPCN Same as JMPNC

Start: LD IN1

JMPC TheRest (\* Jump to "TheRest" if IN1 is TRUE \*)

LD IN2 (\* these three instructions are not executed \*)

ST Q2 (\* if IN1 is TRUE \*)

JMP TheEnd (\* unconditional jump to "TheEnd" \*)

TheRest: LD IN3

ST Q3

TheEnd:

## **RETURN RET RETC RETNC RETCN**

### Labels | Jumps

*Statement* - Jump to the end of the program.

#### **Remarks**

The "RETURN" statement jumps to the end of the program. In FBD language, the return statement is represented by the "<RETURN>" symbol. The input of the symbol must be connected to a valid boolean signal. The jump is performed only if the input is TRUE. In LD language, the "<RETURN>" symbol is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE. In IL language, RET, RETC, RETCN and RETNC instructions are used.

When used within an action block of a SFC step, the RETURN statement jumps to the end of the action block.

#### **ST Language**

```
IF NOT bEnable THEN
```

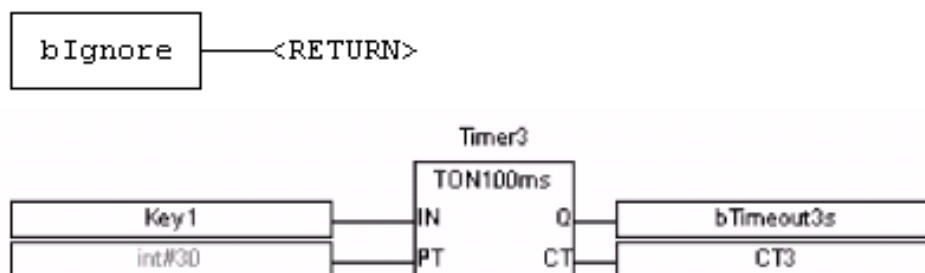
```
    RETURN;
```

```
END_IF;
```

(\* the rest of the program will not be executed  
if bEnabled is FALSE \*)

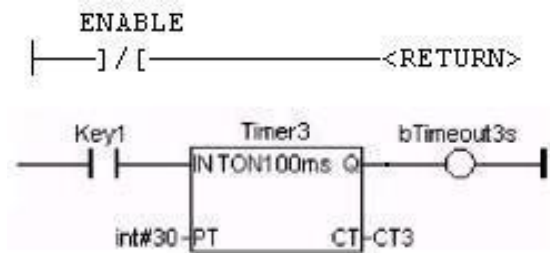
#### **FBD Language**

(\* In this example the TON100ms block will not be called if bIgnore is TRUE \*)



#### **LD Language**

(\* In this example the second rung will be skipped if ENABLE is FALSE \*)



### IL Language

Below is the meaning of possible instructions:

RET Jump to the end always

RETC Jump to the end if the current result is TRUE

RETNC Jump to the end if the current result is FALSE

RETCN Same as RETNC

Start: LD IN1

RETC (\* Jump to the end if IN1 is TRUE \*)

LD IN2 (\* these instructions are not executed \*)

ST Q2 (\* if IN1 is TRUE \*)

RET (\* Jump to the end unconditionally \*)

LD IN3 (\* these instructions are never executed \*)

ST Q3

## **IF THEN ELSE ELSIF END\_IF**

WHILE | REPEAT | FOR | CASE | EXIT

*Statement* - Conditional execution of statements.

### **Syntax**

```
IF <BOOL expression> THEN
  <statements>
ELSIF <BOOL expression> THEN
  <statements>
ELSE
  <statements>
END_IF;
```

### **Remarks**

The IF statement is available in ST only. The execution of the statements is conditioned by a boolean expression. ELSIF and ELSE statements are optional. There can be several ELSIF statements.

### **ST Language**

```
(* simple condition *)
IF bCond THEN
  Q1 := IN1;
  Q2 := TRUE;
END_IF;
```

```
(* binary selection *)
IF bCond THEN
  Q1 := IN1;
  Q2 := TRUE;
ELSE
  Q1 := IN2;
  Q2 := FALSE;
END_IF;
```

```
(* enumerated conditions *)
IF bCond1 THEN
  Q1 := IN1;
ELSIF bCond2 THEN
  Q1 := IN2;
ELSIF bCond3 THEN
  Q1 := IN3;
ELSE
  Q1 := IN4;
END_IF;
```

**FBD Language**

*Not available*

**LD Language**

*Not available*

**IL Language**

*Not available*

## **WHILE DO END\_WHILE**

IF | REPEAT | FOR | CASE | EXIT

*Statement* - Repeat a list of statements.

### **Syntax**

```
WHILE <BOOL expression> DO
  <statements>
END_WHILE ;
```

### **Remarks**

The statements between "DO" and "END\_WHILE" are executed while the boolean expression is TRUE. The condition is evaluated *before* the statements are executed. If the condition is FALSE when WHILE is first reached, statements are never executed.

**Warning:** Loop instructions may lead to infinite loops that block the target cycle. Never test the state of an input in the condition as the input will not be refreshed before the next cycle.

### **ST Language**

```
iPos := 0;
WHILE iPos < iMax DO
  MyArray[iPos] := 0;
  iNbCleared := iNbCleared + 1;
END_WHILE;
```

### **FBD Language**

*Not available*

### **LD Language**

*Not available*

### **IL Language**

*Not available*

## **REPEAT UNTIL END\_REPEAT**

IF | WHILE | FOR | CASE | EXIT

*Statement* - Repeat a list of statements.

### **Syntax**

```
REPEAT
  <statements>
UNTIL <BOOL expression> END_REPEAT;
```

### **Remarks**

The statements between "REPEAT" and "UNTIL" are executed until the boolean expression is TRUE. The condition is evaluated *after* the statements are executed. Statements are executed at least once.

**Warning:** Loop instructions may lead to infinite loops that block the target cycle. Never test the state of an input in the condition as the input will not be refreshed before the next cycle.

### **ST Language**

```
iPos := 0;
REPEAT
  MyArray[iPos] := 0;
  iNbCleared := iNbCleared + 1;
  iPos := iPos + 1;
UNTIL iPos = iMax END_REPEAT;
```

### **FBD Language**

*Not available*

### **LD Language**

*Not available*

### **IL Language**

*Not available*



## **FOR TO BY END\_FOR**

IF | WHILE | REPEAT | CASE | EXIT

*Statement* - Iteration of statement execution.

### **Syntax**

```
FOR <index> := <minimum> TO <maximum> BY <step> DO
    <statements>
END_FOR;
```

*index* = DINT internal variable used as index

*minimum* = DINT expression: initial value for *index*

*maximum* = DINT expression: maximum allowed value for *index*

*step* = DINT expression: increasing step of *index* after each iteration (default is 1)

### **Remarks**

The "BY <step>" statement can be omitted. The default value for the step is 1.

### **ST Language**

```
iArrayDim := 10;
```

(\* resets all items of the array to 0 \*)

```
FOR iPos := 0 TO (iArrayDim - 1) DO
    MyArray[iPos] := 0;
END_FOR;
```

(\* set all items with odd index to 1 \*)

```
FOR iPos := 1 TO 9 BY 2 DO
    MyArray[iPos] := 1;
END_FOR;
```

### **FBD Language**

*Not available*

### **LD Language**

*Not available*

### **IL Language**

*Not available*

## **CASE OF ELSE END\_CASE**

IF | WHILE | REPEAT | FOR | EXIT

*Statement* - switch between enumerated statements.

### **Syntax**

```
CASE <DINT expression> OF
<value> :
  <statements>
<value> , <value> :
  <statements>;
<value> .. <value> :
  <statements>;
ELSE
  <statements>
END_CASE;
```

### **Remarks**

All enumerated values correspond to the evaluation of the DINT expression and are possible cases in the execution of the statements. The statements specified after the ELSE keyword are executed if the expression takes a value that is not enumerated in the switch. For each case, you must specify either a value, or a list of possible values separated by comas (",") or a range of values specified by a " min .. max" interval. You must enter space characters before and after the ".." separator.

### **ST Language**

```
(* this example check first prime numbers *)
CASE iNumber OF
0 :
  Alarm := TRUE;
  AlarmText := '0 gives no result';
1 .. 3, 5 :
  bPrime := TRUE;
4, 6 :
  bPrime := FALSE;
ELSE
  Alarm := TRUE;
  AlarmText := 'I don't know after 6 !';
END_CASE;
```

### **FBD Language**

*Not available*

**LD Language**

*Not available*

**IL Language**

*Not available*

## **EXIT**

IF | WHILE | REPEAT | FOR | CASE

*Statement* - Exit from a loop statement

### **Remarks**

The EXIT statement indicates that the current loop (WHILE, REPEAT or FOR) must be finished. The execution continues after the END\_WHILE, END\_REPEAT or END\_FOR keyword or the loop where the EXIT is. EXIT quits only one loop and cannot be used to exit at the same time several levels of nested loops.

**Warning:** loop instructions may lead to infinite loops that block the target cycle.

### **ST Language**

(\* this program searches for the first non null item of an array \*)

iFound = -1; (\* means: not found \*)

FOR iPos := 0 TO (iArrayDim - 1) DO

    IF iPos <> 0 THEN

        iFound := iPos;

        EXIT;

    END\_IF;

END\_FOR;

### **FBD Language**

*Not available*

### **LD Language**

*Not available*

### **IL Language**

*Not available*

## **Function Blocks**

### **Function Blocks**

The following topics detail the set of programming features and standard blocks:

- Boolean Operations
- Arithmetic Operations
- Standard Operations
- Comparison Operations
- Mathematical Operations
- Advanced Operations
- Register Operations
- Conversion Operations
- String Operations
- CANOpen Operations
- Screen Operations
- Serial Operations
- Removable Media Operations
- Counter Operations
- Time and Date Operations
- Move Operations
- PID Operations
- Network Operations
- Floating PID Operations
- Timer Counter Operations

## Boolean Operations

### *Boolean Operations*

Below are the standard operators for managing booleans:

Boolean AND	performs a boolean AND
Boolean OR	performs a boolean OR
XOR	performs an exclusive OR
NOT	performs a boolean negation of its input
S	force a boolean output to TRUE
R	force a boolean output to FALSE

Below are the available blocks for managing boolean signals:

Reset Dominant	reset dominant bistable
Bistable	set dominant bistable
Set Dominant Bistable	rising pulse detection
Rising Pulse Detection	falling pulse detection
Falling Pulse	
Detection	

## **AND ANDN &**

*Operator* - Performs a logical AND of all inputs.

### **Inputs**

IN1 : BOOL First boolean input

IN2 : BOOL Second boolean input

### **Outputs**

Q : BOOL Boolean AND of all inputs

### **Truth table**

IN1	IN2	Q
0	0	0
0	1	0
1	0	0
1	1	1

### **Remarks**

In FBD language, the block may have up to 255 inputs. The block is called "&" in FBD language. In LD language, an AND operation is represented by serialized contacts. In IL language, the AND instruction performs a logical AND between the current result and the operand. The current result must be boolean. The ANDN instruction performs an AND between the current result and the boolean negation of the operand. In ST and IL languages, "&" can be used instead of "AND".

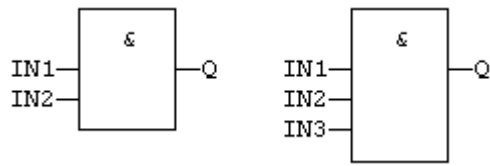
### **ST Language**

Q := IN1 AND IN2;

Q := IN1 & IN2 & IN3;

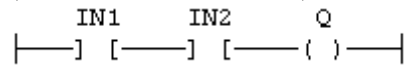
### **FBD Language**

(\* the block may have up to 255 inputs \*)



### LD Language

(\* serialized contacts \*)



### IL Language

Op1: LD IN1

& IN2 (\* "&" or "AND" can be used \*)

ST Q (\* Q is equal to: IN1 AND IN2 \*)

Op2: LD IN1

AND IN2

&N IN3 (\* "&N" or "ANDN" can be used \*)

ST Q (\* Q is equal to: IN1 AND IN2 AND (NOT IN3) \*)

### See Also

OR XOR NOT



## ***F\_TRIG***

*Function Block* - Falling pulse detection

### **Inputs**

CLK : BOOL Boolean signal

### **Outputs**

Q : BOOLTRUE when the input changes from TRUE to FALSE

### **Truth table**

CLK	CLK <i>prev</i>	Q
0	0	0
0	1	1
1	0	0
1	1	0

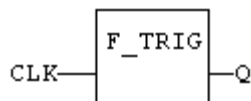
### **ST Language**

(\* MyTrigger is declared as an instance of F\_TRIG function block \*)

MyTrigger (CLK);

Q := MyTrigger.Q;

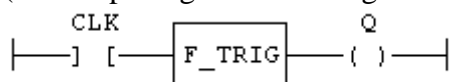
### **FBD Language**



### **LD Language**

(\* The output signal is activated for one execution cycle, every time the input signal goes OFF \*)

(\* the input signal is the rung - the rung is the output \*)



**IL Language**

(\* MyTrigger is declared as an instance of F\_TRIG function block \*)

Op1: CAL MyTrigger (CLK)

LD MyTrigger.Q

ST Q

**See also**

R\_TRIG

OR ORN

*Operator* - Performs a logical OR of all inputs.

Inputs

IN1 : BOOL First boolean input

IN2 : BOOL Second boolean input

### Outputs

Q : BOOL Boolean OR of all inputs

### Truth table

IN1	IN2	Q
0	0	0
0	1	1
1	0	1
1	1	1

### Remarks

In FBD language, the block may have up to 255 inputs. The block is called ">=1" in FBD language. In LD language, an OR operation is represented by contacts in parallel. In IL language, the OR instruction performs a logical OR between the current result and the operand. The current result must be boolean. The ORN instruction performs an OR between the current result and the boolean negation of the operand.

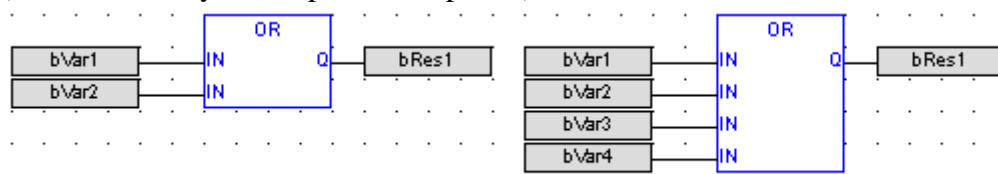
### ST Language

Q := IN1 OR IN2;

Q := IN1 OR IN2 OR IN3;

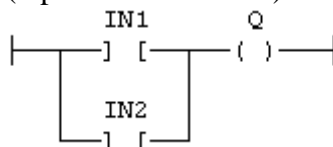
### FBD Language

(\* the block may have up to 255 inputs \*)



### LD Language

(\* parallel contacts \*)



### IL Language

Op1: LD IN1

OR IN2

ST Q (\* Q is equal to: IN1 OR IN2 \*)

Op2: LD IN1  
    ORN IN2  
    ST Q (\* Q is equal to: IN1 OR (NOT IN2) \*)

***See also***

AND XOR NOT

## **R\_TRIG**

*Function Block - Rising pulse detection*

### **Inputs**

CLK : BOOL Boolean signal

### **Outputs**

Q : BOOL TRUE when the input changes from FALSE to TRUE

### **Truth table**

CLK	CLK <i>prev</i>	Q
0	0	0
0	1	0
1	0	1
1	1	0

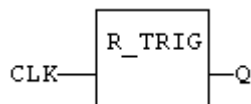
### **ST Language**

(\* MyTrigger is declared as an instance of R\_TRIG function block \*)

MyTrigger (CLK);

Q := MyTrigger.Q;

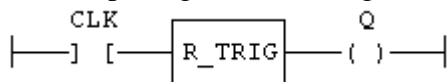
### **FBD Language**



### **LD Language**

(\* The output signal is activated for one execution cycle, every time the input signal goes ON \*)

(\* the input signal is the rung - the rung is the output \*)



### **IL Language**

(\* MyTrigger is declared as an instance of R\_TRIG function block \*)

Op1: CAL MyTrigger (CLK)

LD MyTrigger.Q

ST Q

### **See also**

F\_TRIG

## RS

*Function Block* - Reset dominant bistable.

### Inputs

SET : BOOL Condition for forcing to TRUE

RESET1 : BOOL Condition for forcing to FALSE (highest priority command)

### Outputs

Q1 : BOOL Output to be forced

### Truth table

SET	RESET1	Q1 <sub>prev</sub>	Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

### Remarks

The output is unchanged when both inputs are FALSE. When both inputs are TRUE, the output is forced to FALSE (reset dominant).

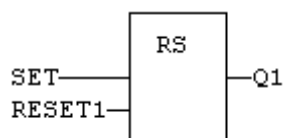
### ST Language

(\* MyRS is declared as an instance of RS function block \*)

MyRS (SET, RESET1);

Q1 := MyRS.Q1;

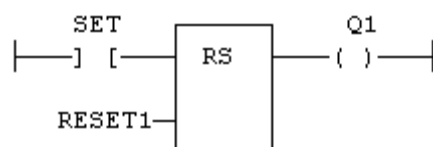
### FBD Language



### LD Language

(\* RESET command is the dominant command in case of both the commands being active simultaneously\*)

(\* the SET command is the rung - the rung is the output \*)



**IL Language**

(\* MyRS is declared as an instance of RS function block \*)

Op1: CAL MyRS (SET, RESET1)

LD MyRS.Q1

ST Q1

**See also**

R S SR

## SR

*Function Block* - Set dominant bistable.

### Inputs

SET1 : BOOL Condition for forcing to TRUE (highest priority command)

RESET : BOOL Condition for forcing to FALSE

### Outputs

Q1 : BOOL Output to be forced

### Truth table

SET1	RESET	Q1 <sub>prev</sub>	Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

### Remarks

The output is unchanged when both inputs are FALSE. When both inputs are TRUE, the output is forced to TRUE (set dominant).

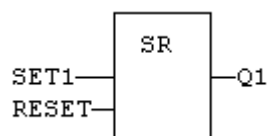
### ST Language

(\* MySR is declared as an instance of SR function block \*)

MySR (SET1, RESET);

Q1 := MySR.Q1;

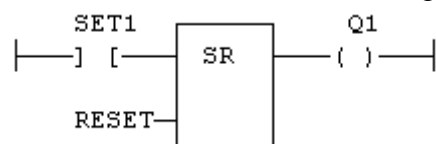
### FBD Language



### LD Language

(\*SET command is the dominant command in case of both the commands being active simultaneously\*)

(\* the SET1 command is the rung - the rung is the output \*)





**IL Language**

(\* MySR is declared as an instance of SR function block \*)

Op1: CAL MySR (SET1, RESET)

LD MySR.Q1

ST Q1

**See also**

R S RS

## **XOR XORN**

*Operator* - Performs an exclusive OR of all inputs.

### **Inputs**

IN1 : BOOL First boolean input

IN2 : BOOL Second boolean input

### **Outputs**

Q : BOOL Exclusive OR of all inputs

### **Truth table**

IN1	IN2	Q
0	0	0
0	1	1
1	0	1
1	1	0

### **Remarks**

The block is called "=1" in FBD and LD languages. In IL language, the XOR instruction performs an exclusive OR between the current result and the operand. The current result must be boolean. The XORN instruction performs an exclusive between the current result and the boolean negation of the operand.

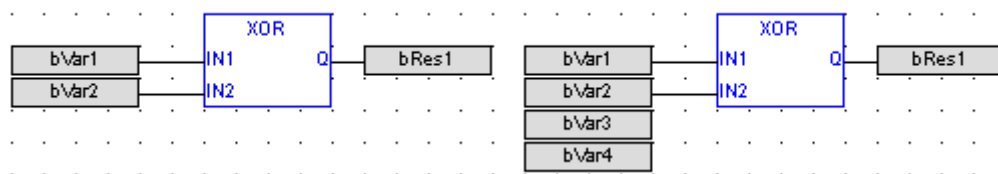
### **ST Language**

Q := IN1 XOR IN2;

Q := IN1 XOR IN2 XOR IN3;

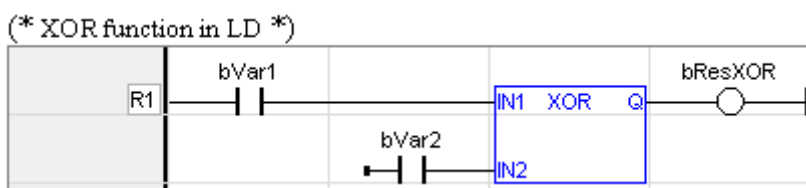
### **FBD Language**

(\* the block may have up to 255 inputs \*)



### **LD Language**

(\* First input is the rung. The rung is the output \*)



### **IL Language**

Op1: LD IN1  
XOR IN2  
ST Q (\* Q is equal to: IN1 XOR IN2 \*)  
Op2: LD IN1  
XORN IN2  
ST Q (\* Q is equal to: IN1 XOR (NOT IN2) \*)

**See also**

AND OR NOT

## S

*Operator* - Force a boolean output to TRUE.

### Inputs

SET : BOOL Condition

### Outputs

Q : BOOL Output to be forced

### Truth table

SET	Q <sub>prev</sub>	Q
0	0	0
0	1	1
1	0	1
1	1	1

### Remarks

S and R operators are available as standard instructions in the IL language. In LD languages they are represented by (S) and (R) coils. In FBD language, you can use (S) and (R) coils, but you should prefer RS and SR function blocks. Set and reset operations are not available in ST language.

### ST Language

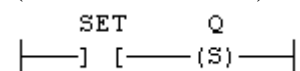
*Not available.*

### FBD Language

*Prefer the use of RS or SR function blocks.*

### LD Language

(\* use of "S" coil \*)



### IL Language

Op1: LD SET

S Q (\* Q is forced to TRUE if SET is TRUE \*)  
(\* Q is unchanged if SET is FALSE \*)

### See also

R RS SR

## R

*Operator* - Force a boolean output to FALSE.

### Inputs

RESET : BOOL Condition

### Outputs

Q : BOOL Output to be forced

### Truth table

RESET	Q <sub>prev</sub>	Q
0	0	0
0	1	1
1	0	0
1	1	0

### Remarks

S and R operators are available as standard instructions in the IL language. In LD languages they are represented by (S) and (R) coils. In FBD language, you can use (S) and (R) coils, but you should prefer RS and SR function blocks. Set and reset operations are not available in ST language.

### ST Language

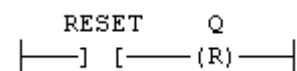
*Not available.*

### FBD Language

*Prefer the use of RS or SR function blocks.*

### LD Language

(\* use of "R" coil \*)



### IL Language

Op1: LD RESET

R Q (\* Q is forced to FALSE if RESET is TRUE \*)  
(\* Q is unchanged if RESET is FALSE \*)

### See also

S RS SR

## **Arithmetic Operations**

### ***Arithmetic Operations***

Below are the standard operators that perform arithmetic operations:

Add	addition
Subtract	subtraction
Multiply	multiplication
Divide	division

## \* **MUL**

*Operator* - Performs a multiplication of all inputs.

### Inputs

IN1 : ANY (numeric) First input

IN2 : ANY (numeric) Second input

### Outputs

Q : ANY\_NUM Result:  $IN1 * IN2$

### Remarks

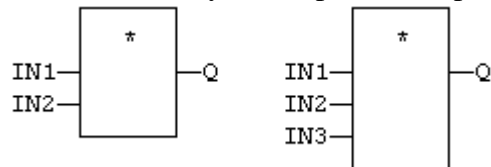
All inputs and the output must have the same type. In FBD language, the block may have up to 255 inputs. In LD language, the EN signal enables the operation, and the ENO keeps the same value as the EN. In IL language, the MUL instruction performs a multiplication between the current result and the operand. The current result and the operand must have the same type.

### ST Language

Q := IN1 \* IN2;

### FBD Language

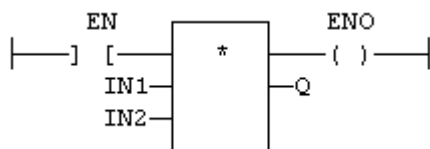
(\* the block may have up to 255 inputs \*)



### LD Language

(\* The multiplication is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### IL Language

Op1: LD IN1

MUL IN2

ST Q (\* Q is equal to:  $IN1 * IN2$  \*)

Op2: LD IN1

MUL IN2

MUL IN3

ST Q (\* Q is equal to:  $IN1 * IN2 * IN3$  \*)

### See also

+ - /

## **+ ADD**

*Operator* - Performs an addition of all inputs.

### **Inputs**

IN1 : ANY (numeric) First input

IN2 : ANY (numeric) Second input

### **Outputs**

Q : ANY      Result:  $IN1 + IN2$

### **Remarks**

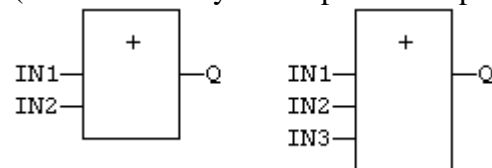
All inputs and the output must have the same type. In FBD language, the block may have up to 255 inputs. In LD language, the EN signal enables the operation, and the ENO keeps the same value as the EN. In IL language, the ADD instruction performs an addition between the current result and the operand. The current result and the operand must have the same type.

### **ST Language**

$Q := IN1 + IN2;$

### **FBD Language**

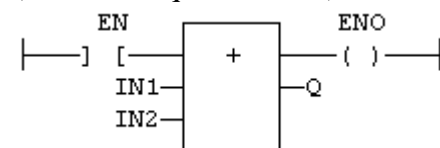
(\* the block may have up to 255 inputs \*)



### **LD Language**

(\* The addition is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### **IL Language**

Op1: LD IN1

ADD IN2

ST Q (\* Q is equal to:  $IN1 + IN2$  \*)

Op2: LD IN1

ADD IN2

ADD IN3

ST Q (\* Q is equal to:  $IN1 + IN2 + IN3$  \*)

### **See also**

- \* /



- SUB

*Operator* - Performs a subtraction of inputs.

### **Inputs**

IN1 : ANY (numeric) / TIME First input

IN2 : ANY (numeric) / TIME Second input

### **Outputs**

Q : ANY\_NUM / TIME Result: IN1 - IN2

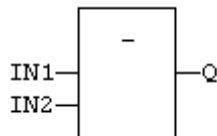
### **Remarks**

All inputs and the output must have the same type. In LD language, the EN signal enables the operation, and the ENO keeps the same value as the EN. In IL language, the SUB instruction performs a subtraction between the current result and the operand. The current result and the operand must have the same type.

### **ST Language**

Q := IN1 - IN2;

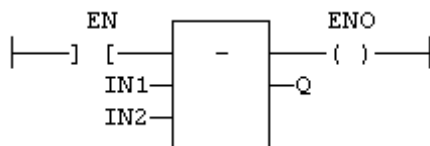
### **FBD Language**



### **LD Language**

(\* The subtraction is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### **IL Language**

Op1: LD IN1

SUB IN2

ST Q (\* Q is equal to: IN1 - IN2 \*)

Op2: LD IN1

SUB IN2

SUB IN3

ST Q (\* Q is equal to: IN1 - IN2 - IN3 \*)

### **See also**

+ \* /

/ DIV

*Operator* - Performs a division of inputs.

### **Inputs**

IN1 : ANY (numeric) First input

IN2 : ANY (numeric) Second input

### **Outputs**

Q : ANY\_NUM Result: IN1 / IN2

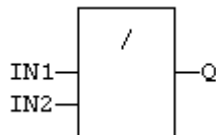
### **Remarks**

All inputs and the output must have the same type. In LD language, the input rung EN signal enables the operation, and the ENO keeps the same value as the EN. In IL language, the DIV instruction performs a division between the current result and the operand. The current result and the operand must have the same type.

### **ST Language**

Q := IN1 / IN2;

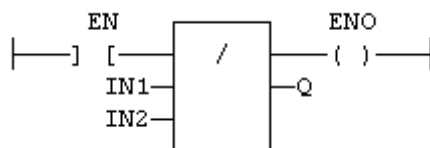
### **FBD Language**



### **LD Language**

(\* The division is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### **IL Language**

Op1: LD IN1

    DIV IN2

    ST Q (\* Q is equal to: IN1 / IN2 \*)

Op2: LD IN1

    DIV IN2

    DIV IN3

    ST Q (\* Q is equal to: IN1 / IN2 / IN3 \*)

### **See also**

+ - \*

## **Standard Operations**

### ***Standard Operations***

Below are the standard functions for managing standard operations:

Copy 1 Gain

Negation

Boolean Negation

## Copy 1 Gain

Operator - variable assignment.

### Inputs

IN : ANY Any variable or complex expression

### Outputs

Q : ANY Forced variable

### Remarks

The output variable and the input expression must have the same type. The forced variable cannot have the "read only" attribute. In LD and FBD languages, the "1" block is available to perform a "1 gain" data copy. In LD language, the EN enables the assignment, and the ENO keeps the state of the EN. In IL language, the LD instruction loads the first operand, and the ST instruction stores the current result into a variable. The current result and the operand of ST must have the same type. Both LD and ST instructions can be modified by "N" in case of a boolean operand for performing a boolean negation.

### ST Language

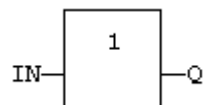
Q := IN; (\* copy IN into variable Q \*)

Q := (IN1 + (IN2 / IN 3)) \* IN4; (\* assign the result of a complex expression \*)

result := SIN (angle); (\* assign a variable with the result of a function \*)

time := MyTon.ET; (\* assign a variable with an output parameter of a function block \*)

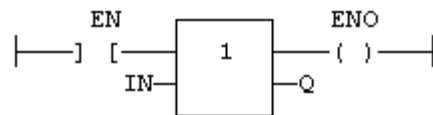
### FBD Language



### LD Language

(\* The copy is executed only if EN is TRUE \*)

(\* ENO has the same value as EN \*)



### IL Language

LD IN (IN & Q assign as INT data type)

IN1

ST Q

### See also

Parenthesis

NEG -

*Operator* - Performs an integer negation of the input.

### **Inputs**

IN : DINT Integer value

### **Outputs**

Q : DINT Integer negation of the input

### **Truth table (examples)**

IN	Q
0	0
1	-1
-123	123

### **Remarks**

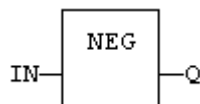
In FBD and LD language, the block "NEG" can be used. In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. This feature is not available in IL language. In ST language, "-" can be followed by a complex boolean expression between parenthesis.

### **ST Language**

Q := -IN;

Q := - (IN1 + IN2);

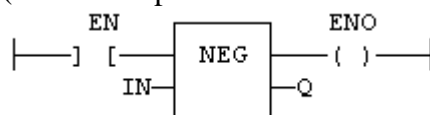
### **FBD Language**



### **LD Language**

(\* The negation is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

LD-IN1 \*IN1 & Q assign as INT data types)

NEG:

ST Q

NOT

*Operator* - Performs a boolean negation of the input.

### Inputs

IN : BOOL Boolean value

### Outputs

Q : BOOL Boolean negation of the input

### Truth table

IN	Q
0	1
1	0

### Remarks

In FBD language, the block "NOT" can be used. Alternatively, you can use a link terminated by a "o" negation. In LD language, negated contacts and coils can be used. In IL language, the "N" modifier can be used with instructions LD, AND, OR, XOR and ST. It represents a negation of the operand. In ST language, NOT can be followed by a complex boolean expression between parenthesis.

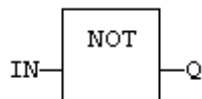
### ST Language

Q := NOT IN;

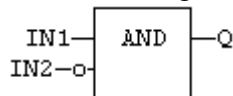
Q := NOT (IN1 OR IN2);

### FBD Language

(\* explicit use of the "NOT" block \*)

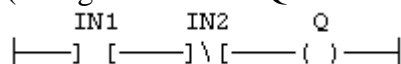


(\* use of a negated link: Q is IN1 AND NOT IN2 \*)

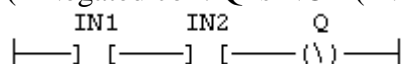


### LD Language

(\* Negated contact: Q is: IN1 AND NOT IN2 \*)



(\* Negated coil: Q is NOT (IN1 AND IN2) \*)



### IL Language

Op1: LDN IN1  
OR IN2

ST Q (\* Q is equal to: (NOT IN1) OR IN2 \*)  
Op2: LD IN1  
AND IN2  
STN Q (\* Q is equal to: NOT (IN1 AND IN2) \*)

***See also***

AND OR XOR

## Comparison Operations

### *Comparison Operations*

Below are the standard operators and blocks that perform comparisons:

Less	Than	less than
Greater	Than	greater than
Less	Than or	less or equal
Equal		greater or equal
Greater	Than or	is equal
Equal		is not equal
Equal		is within or out of range
Not	Equal	
Bounds Test		



< LT

*Operator* - Test if first input is less than second input.

### **Inputs**

IN1 : ANY First input

IN2 : ANY Second input

### **Outputs**

Q : BOOL TRUE if IN1 < IN2

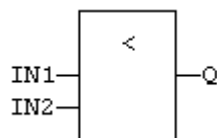
### **Remarks**

IN1 & IN2 must have the same data type. In LD language, the EN signal enables the operation, and the ENO is the result of the comparison. In IL language, the LT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

### **ST Language**

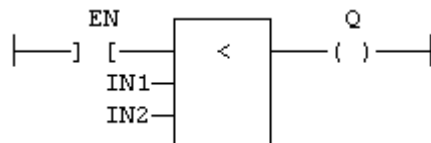
Q := IN1 < IN2;

### **FBD Language**



### **LD Language**

(\* The comparison is executed only if EN is TRUE \*)



### **IL Language**

Op1: LD IN1

LT IN2

ST Q (\* Q is true if IN1 < IN2 \*)

### **See also**

> >= <= = <>

**<= LE**

*Operator* - Test if first input is less than or equal to second input.

### **Inputs**

IN1 : ANY First input

IN2 : ANY Second input

### **Outputs**

Q : BOOL TRUE if IN1 <= IN2

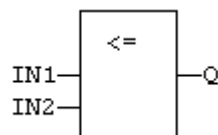
### **Remarks**

Both inputs must have the same type. In LD language, the the EN signal enables the operation, and the ENO is the result of the comparison. In IL language, the LE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

### **ST Language**

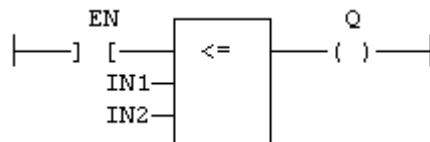
Q := IN1 <= IN2;

### **FBD Language**



### **LD Language**

(\* The comparison is executed only if EN is TRUE \*)



### **IL Language**

Op1: LD IN1

LE IN2

ST Q (\* Q is true if IN1 <= IN2 \*)

### **See also**

> < >= = <>

<> NE

*Operator* - Test if first input is not equal to second input.

### **Inputs**

IN1 : ANY First input

IN2 : ANY Second input

### **Outputs**

Q : BOOL TRUE if IN1 is not equal to IN2

### **Remarks**

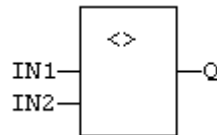
Both inputs must have the same type. In LD language, the input rung EN signal enables the operation, and the ENO is the result of the comparison. In IL language, the NE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Equality comparisons cannot be used with TIME variables. Because the timer actually has the resolution of the target cycle and test may be unsafe as some values may never be reached

### **ST Language**

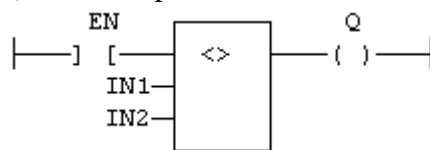
Q := IN1 <> IN2;

### **FBD Language**



### **LD Language**

(\* The comparison is executed only if EN is TRUE \*)



### **IL Language**

Op1: LD IN1

NE IN2

ST Q (\* Q is true if IN1 is not equal to IN2 \*)

### **See also**

> < >= <= =

= EQ

*Operator* - Test if first input is equal to second input.

### **Inputs**

IN1 : ANY First input

IN2 : ANY Second input

### **Outputs**

Q : BOOL TRUE if IN1 = IN2

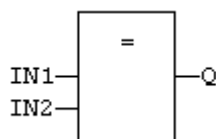
### **Remarks**

Both inputs must have the same type. In LD language, the input rung EN signal enables the operation, and the ENO is the result of the comparison. In IL language, the EQ instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type. Equality comparisons cannot be used with TIME variables. Because the timer actually has the resolution of the target cycle and test may be unsafe as some values may never be reached.

### **ST Language**

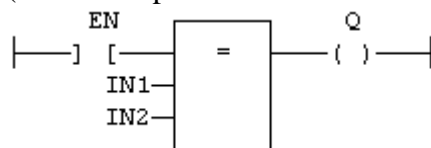
Q := IN1 = IN2;

### **FBD Language**



### **LD Language**

(\* The comparison is executed only if EN is TRUE \*)



### **IL Language**

Op1: LD IN1

EQ IN2

ST Q (\* Q is true if IN1 = IN2 \*)

### **See also**

> < >= <= <>

> GT

*Operator* - Test if first input is greater than second input.

### **Inputs**

IN1 : ANY First input

IN2 : ANY Second input

### **Outputs**

Q : BOOL TRUE if  $IN1 > IN2$

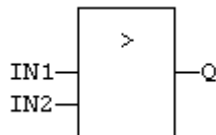
### **Remarks**

Both inputs must have the same type. In LD language, the input rung EN signal enables the operation, and the ENO is the result of the comparison. In IL language, the GT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

### **ST Language**

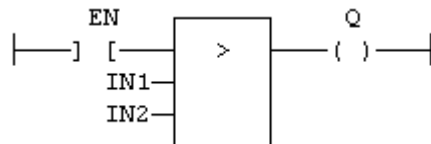
Q := IN1 > IN2;

### **FBD Language**



### **LD Language**

(\* The comparison is executed only if EN is TRUE \*)



### **IL Language**

Op1: LD IN1

GT IN2

ST Q (\* Q is true if  $IN1 > IN2$  \*)

### **See also**

< >= <= = <>

**>= GE**

- Test if first input is greater than or equal to second input.

### **Inputs**

IN1 : ANY First input

IN2 : ANY Second input

### **Outputs**

Q : BOOL TRUE if IN1 >= IN2

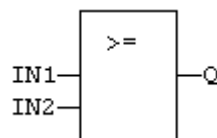
### **Remarks**

Both inputs must have the same type. In LD language, the input rung EN signal enables the operation, and the ENO is the result of the comparison. In IL language, the GE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

### **ST Language**

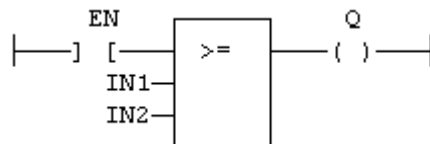
Q := IN1 >= IN2;

### **FBD Language**



### **LD Language**

(\* The comparison is executed only if EN is TRUE \*)



### **IL Language**

Op1: LD IN1

GE IN2

ST Q (\* Q is true if IN1 >= IN2 \*)

### **See also**

> < <= = <>

## **LIM**

*Operator* – This block determines if the input (IN) value is numerically in the range defined by the Low and High.

### **Inputs**

LOW: This is the lower range for the Input IN. (TYPE : ANY)

IN: The Input which is checked to lie between the Low & High ranges specified at the Inputs. (TYPE : ANY)

HIGH: This is the Higher range for the Input IN. (TYPE : ANY)

### **Outputs**

Q : The Output is true when input IN is in between the LOW/HIGH range. (TYPE: BOOL)

### **Remarks**

#### **If Low <= High:**

This function passes power if the input is in between Low and High (inclusive) range. For example, if Low = 10 and High = 100 when the INPUT is between 10 and 100 the function passes power. If the input is 9 or lower OR 101 or higher this function would **not** pass power.

#### **If Low > High:**

This function passes power if the input is **outside** the range of Low and High (exclusive).

For example, if Low = 100 and High = 10 when the INPUT is between 11 and 99 the function does **not** pass power. If the input is 10 or lower OR 100 or higher this function will pass power.

### **ST Language**

#### **Example:**

LIM (LOW, IN, HIGH)

Q: = LIM. Q;

### **FBD Language**

### **LD Language**

### **IL Language**

#### Example:

Op1: CAL LIM (LOW, IN, HIGH);

LD LIM. Q

ST Q

## Mathematical Operations

### *Mathematical Operations*

Below are the standard functions that perform mathematical calculation:

Absolute Value	absolute value
Arc-Cosine	
Arc-Sine	
Arc-Tangent	
Cosine	
Degrees to	
Radians	
Power of e	
Exponent of X	
Base 10	logarithm
Logarithm	
Logarithm	
Radian To	
Degrees	square root
Sine	
Modulo	
Square Root	
Tangent	
Trigonometric	
Operations	



## ABS

*Function* - Returns the absolute value of the input.

### Inputs

IN : REAL Real value

### Outputs

Q : REAL/LREAL Result: absolute value of IN

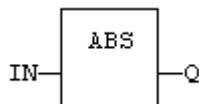
### Remarks

In LD language, the operation is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL, the input must be loaded in the current result before calling the function.

### ST Language

Q := ABS (IN);

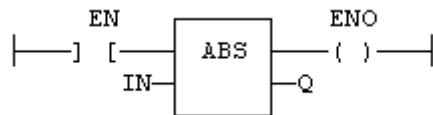
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

Op1: LD IN

ABS

ST Q (\* Q is: ABS (IN) \*)

### See also

LOG SQRT

## ACOS

*Function* - Calculate an arc-cosine.

### Inputs

IN : REAL Real value

### Outputs

Q : REAL Result: arc-cosine of IN

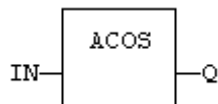
### Remarks

In LD language, the operation is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL, the input must be loaded in the current result before calling the function.

### ST Language

Q := ACOS (IN);

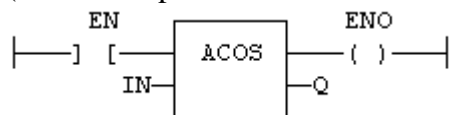
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

Op1: LD IN

ACOS

ST Q (\* Q is: ACOS (IN) \*)

### See also

SIN COS TAN ASIN ATAN

ASIN

*Function* - Calculate an arc-sine.

### **Inputs**

IN : REAL/LREAL Real value

### **Outputs**

Q : REAL/LREAL Result: arc-sine of IN

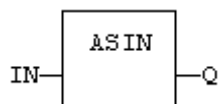
### **Remarks**

In LD language, the operation is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL, the input must be loaded in the current result before calling the function.

### **ST Language**

Q := ASIN (IN);

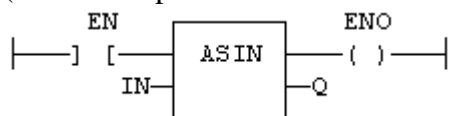
### **FBD Language**



### **LD Language**

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

Op1: LD IN

ASIN

ST (\* Q is: ASIN (IN) \*)

### **See also**

SIN COS TAN ACOS ATAN

## ATAN

*Function* - Calculate an arc-tangent.

### Inputs

IN : REAL Real value

### Outputs

Q : REAL Result: arc-tangent of IN

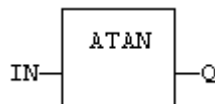
### Remarks

In LD language, the operation is executed only if the EN signal is TRUE. The ENO keeps the same value as the EN. In IL, the input must be loaded in the current result before calling the function.

### ST Language

Q := ATAN (IN);

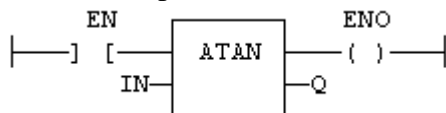
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

Op1: LD IN

ATAN

ST Q (\* Q is: ATAN (IN) \*)

### See also

SIN COS TAN ASIN ACOS

## COS

*Function* - Calculate a cosine.

### Inputs

IN : REAL/LREAL Real value

### Outputs

Q : REAL/LREAL Result: cosine of IN

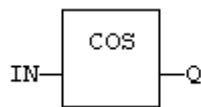
### Remarks

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

### ST Language

Q := COS (IN);

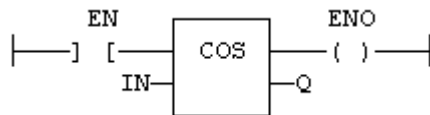
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

Op1: LD IN

COS

ST (\* Q is: COS (IN) \*)

### See also

SIN TAN ASIN ACOS ATAN

DegToRad

*Operator* – Converts Degrees to Radians.

### **Inputs**

IN: Input in Degrees. ( TYPE : REAL)

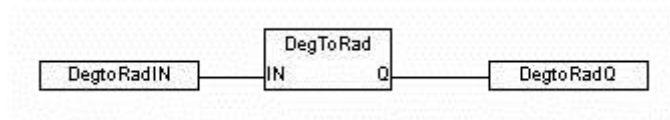
### **Outputs**

Q : Converted input to Radians. ( TYPE : REAL)

### **ST Language**

Q := DegToRad ( IN);

### **FBD Language**



### **LD Language**



### **IL Language**

Op1: LD IN  
    DegToRad  
    ST Q

### **See also**

[RadToDeg](#)

EXT

*Operator* – This function determines the value of  $e$  (the base of natural logarithms) raised to the **IN**'th power and places the result in **Q**.

***Inputs***

IN: Number used for finding the natural log. (TYPE : REAL)

***Outputs***

Q : The result of  $e$  raised to the power of **IN**. (TYPE : REAL)

**Q = EXT(IN)**

***ST Language***

Q = EXT(IN);

***FBD Language***

***LD Language***

***IL Language***

Op1: LD IN  
EXT  
ST Q

***See also***

E  
X  
P  
T

EXPT

*Operator* – This function raises **IN** to the **EXP**'th power and places the result in **Q**

***Inputs***

IN: Number which needs to be raised to **EXP**'th Power (TYPE : REAL)

EXP: Number used for increasing the Input IN to this power (TYPE : REAL)

***Outputs***

Q : The result of **IN** raised to the power of **EXP**. (TYPE : REAL)

**Q = IN^^EXP**

***ST Language***

Q = EXPT(IN, EXP)

***FBD Language***

***LD Language***

***IL Language***

Op1: LD IN  
EXPT EXP  
ST Q

***See also***

E  
X  
T



## LOG10

*Function* – Calculates the logarithm (base 10) of the input.

### Inputs

IN: Input to determine LOG to base 10. ( TYPE : REAL)

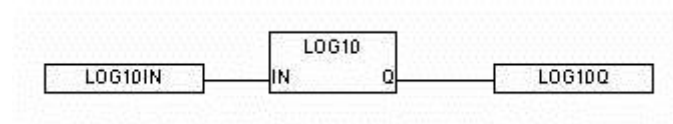
### Outputs

Q : Outputs LOG to base 10 for the Input at IN. (TYPE: REAL)

### ST Language

Q := LOG10(IN);

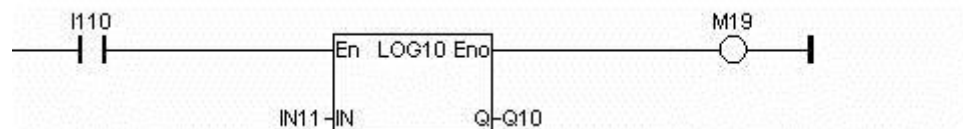
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

```
Op1: LD IN
      LOG10
      ST Q(* Q is: LOG10(IN) *)
```

### See also

LOG

## LOG

*Function* - Calculates the logarithm (base e) of the input.

### Inputs

IN : REAL Real value

### Outputs

Q : REAL Result: logarithm (base e) of IN

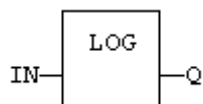
### Remarks

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

### ST Language

Q := LOG (IN);

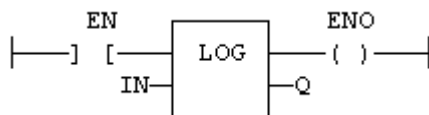
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

Op1: LD IN

LOG

ST Q (\* Q is: LOG (IN) \*)

### See also

ABS POW SQRT

RadToDeg

*Operator* – Converts Radians to Degrees.

### **Inputs**

IN: Input in Radians.( TYPE : REAL)

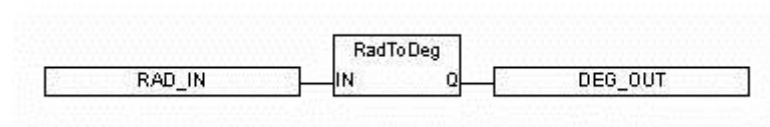
### **Outputs**

Q : Converted input to Degrees. ( TYPE : REAL)

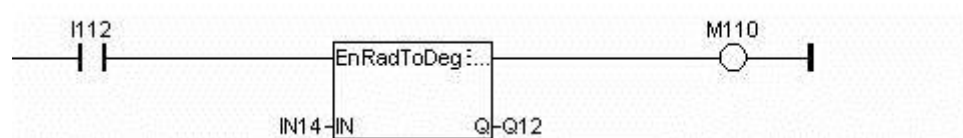
### **ST Language**

Q := RadToDeg(IN);

### **FBD Language**



### **LD Language**



### **IL Language**

Op1: LD IN  
RadToDeg  
ST Q

### **See also**

[DegToRad](#)

## SIN / SINL

*Function* - Calculate a sine.

### Inputs

IN : REAL/LREAL Real value

### Outputs

Q : REAL/LREAL Result: sine of IN

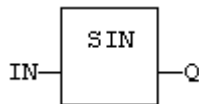
### Remarks

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

### ST Language

Q := SIN (IN);

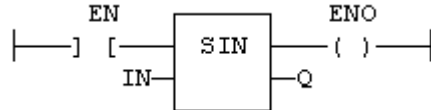
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

Op1: LD IN

SIN

ST Q (\* Q is: SIN (IN) \*)

### See also

COS TAN ASIN ACOS ATAN ATAN2

## MOD

*Function* - Calculation of modulo.

### Inputs

IN : DINT Input value

BASE : DINT Base of the modulo

### Outputs

Q : DINT Modulo: rest of the integer division (IN / BASE)

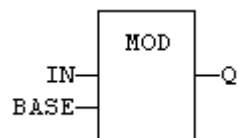
### Remarks

In LD language, the input rung EN signal enables the operation, and the ENO keeps the state of the EN. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

### ST Language

Q := MOD (IN, BASE);

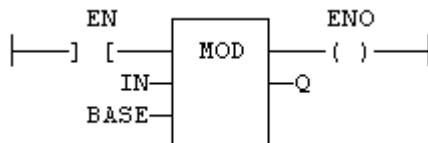
### FBD Language



### LD Language

(\* The comparison is executed only if EN is TRUE \*)

(\* ENO has the same value as EN \*)



### IL Language

Op1: LD IN

MOD BASE

ST Q (\* Q is the rest of integer division: IN / BASE \*)

## SQRT

*Function* - Calculates the square root of the input.

### Inputs

IN : REAL Real value

### Outputs

Q : REAL Result: square root of IN

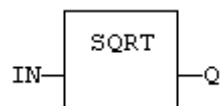
### Remarks

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

### ST Language

Q := Sqrt (IN);

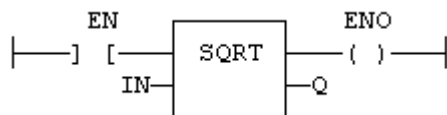
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### IL Language

Op1: LD IN

SQRT

ST Q (\* Q is: Sqrt (IN) \*)

### See also

ABS LOG

TAN

*Function* - Calculate a tangent.

### **Inputs**

IN : REAL Real value

### **Outputs**

Q : REAL Result: tangent of IN

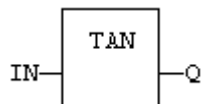
### **Remarks**

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

### **ST Language**

Q := TAN (IN);

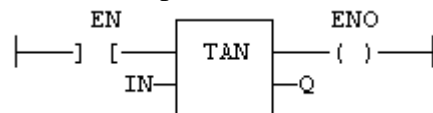
### **FBD Language**



### **LD Language**

(\* The function is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language:**

Op1: LD IN

TAN

ST Q (\* Q is: TAN (IN) \*)

### **See also**

SIN COS ASIN ACOS ATAN

## **Advanced Operations**

### ***Advanced Operations***

Below are the standard operators that perform Advanced operations:

Alarm Handling

Alarm with  
Date/Time Stamp

Set Clock

Real Scaling

Integer Scaling

Indexed Stepper

Move

Stepper Move



## Alarm

*Operator* - Creates an alarm handler from a table of contiguous alarm bits in the [i3](#). Within the table each bit relates to an individual alarm and the status of each alarm is monitored.

### Inputs

CB[ ] : Control Block (TYPE : UINT[])

Each alarm require one 16-bit status register. The registers for multiple alarms are defined in a contiguous block called the Control Block. One bit is written to this register to indicate that the alarm is active. The register also contains sections that indicates the acknowledge and pending status and contains a count for the alarm. By placing the alarm status registers in a section of retentive memory (%R, %M...) the alarm states will be retained through a power cycle.

The following table shows how the bits in the alarm status word (control block) are allocated:

Bits:	16-12	11	10	9	8-1
Definition:	Undefined*	Acknowledge	Pending	Active	Alarm Count

NEXT: (TYPE : BOOL)

When this input transitions from low to high, the next (higher alarm number) pending alarm is shown on the display. If the highest alarm is being displayed, the alarm number is not incremented further.

PREV: (TYPE : BOOL)

When this input transitions from low to high, the previous (lower alarm number) pending alarm is shown on the display. If the lowest alarm is being displayed, the alarm number is not decremented further.

CLEAR: (TYPE : BOOL)

When this input transitions from low to high, the currently displayed alarm is cleared if it has already been acknowledged. If it has not been acknowledged this input has no affect. Once the alarm is cleared, the function block immediately searches for the next active alarm screen to display by searching for the next (higher) alarm status register with a Pending bit set. If an alarm is cleared that is still active, the pending bit will be set and if no other alarm is active will continue to be displayed

ACK: (TYPE : BOOL)

When this input transitions from low to high, the currently displayed alarm is marked as acknowledged. This will set the acknowledge bit in the status register and allow the alarm to be cleared

#SCREEN1: (TYPE : DINT)

This defines the first in a block of screens that will be used to display alarm information.

#COUNT: (TYPE : DINT)

Count is the total number of alarms defined. This number also sets how many registers are used for status registers, how many text screens are reserved for alarm display.

### Remarks

#### 1) The Control Block

**Alarm Count** - This is a BYTE counter that counts how many times an alarm occurs. The count only increments when the pending bit goes from low to high. To count another alarm event the alarm must be acknowledged, cleared and reactivated. When the count reaches a maximum of 255 it no longer changes until reset. This count can be reset by writing directly to this portion of the register using one of the BYTE instructions.

**Active** - This bit is set by the user's ladder program to indicate an alarm condition has occurred. For example, if the alarm is to indicate an over-temperature condition, have the ladder logic perform a compare, then set this bit if the compare indicates the temperature is greater than a setpoint.

**Pending** - This bit is set by the function block when the Active bit is high and is reset through the functions block's Clear operation. If the Active bit is high when Pending is reset, a new alarm will be recognized and Pending will be set immediately.

**Acknowledge** - This bit is set by the function block after a pending alarm has been acknowledged.

## 2) Special Status Bits

Bit 16 of the first status word turns ON when any alarm is pending (but may be acknowledged).

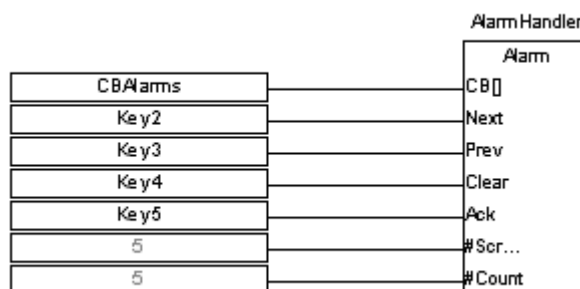
Bit 15 of the first status word turn ON when any alarm is unacknowledged.

## ST Language

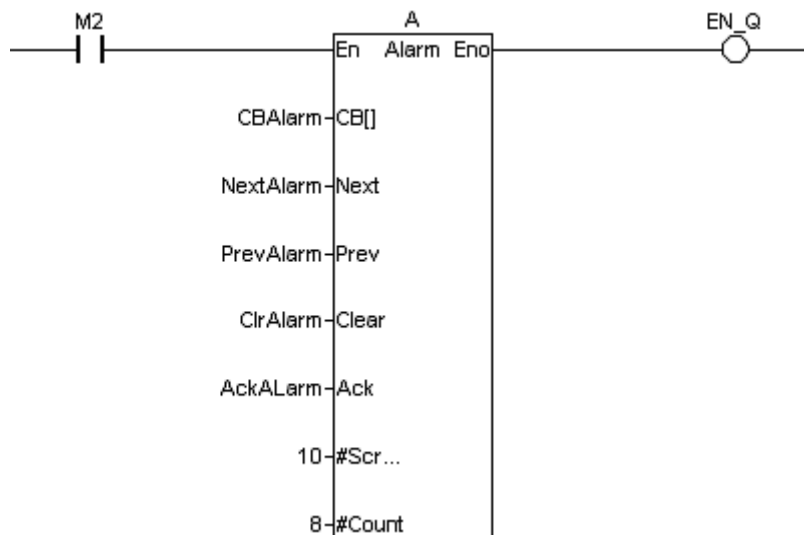
(\* AB is a declared instance of Alarm function block \*)

AB(CB[], NEXT, PREV, CLEAR, ACK, #SCREEN1, #COUNT);

## FBD Language



## LD Language



### IL Language

(\* AB is a declared instance of Alarm function block \*)

OP1 : CAL AB(CB[], NEXT, PREV, CLEAR, ACK, #SCREEN1, #COUNT)

### See also

A  
l  
a  
r  
m  
S  
t  
a  
m  
p

## AlarmStamp

*Operator* - Creates an alarm handler from a table of contiguous alarm bits in the [i3](#). Within the table each bit relates to an individual alarm and the status of each alarm is monitored. In addition a time stamp table is maintained for each alarm indicating the last time the alarm was activated, acknowledged and cleared.

### Inputs

CB[ ] : Control Block (TYPE : UINT[])

Each alarm requires one 16-bit status register. The registers for multiple alarms are defined in a contiguous block called the Control Block. One bit is written to this register to indicate that the alarm is active. The register also contains sections that indicates the acknowledge and pending status and contains a count for the alarm. By placing the alarm status registers in a section of retentive memory (%R, %M...) the alarm states will be retained through a power cycle.

The following table shows how the bits in the alarm status word (control block) are allocated:

Bits:	16-12	11	10	9	8-1
Definition:	Undefined*	Acknowledge	Pending	Active	Alarm Count

NEXT: (TYPE : BOOL)

When this input transitions from low to high, the next(highest alarm number) pending alarm is shown on the display. If the highest alarm is being displayed, the alarm number is not incremented further.

PREV: (TYPE : BOOL)

When this input transitions from low to high, the previous (lower alarm number) pending alarm is shown on the display. If the lowest alarm is being displayed, the alarm number is not decremented further.

CLEAR: (TYPE : BOOL)

When this input transitions from low to high, the currently displayed alarm is cleared if it has already been acknowledged. If it has not been acknowledged this input has no affect. Once the alarm is cleared, the function block immediately searches for the next active alarm screen to display by searching for the next (higher) alarm status register with a Pending bit set. If an alarm is cleared that is still active, the pending bit will be set and if no other alarm is active will continue to be displayed

ACK: (TYPE : BOOL)

When this input transitions from low to high, the currently displayed alarm is marked as acknowledged. This will set the acknowledge bit in the status register and allow the alarm to be cleared.

#SCREEN1 : (TYPE : DINT)

This defines the first in a block of screens that will be used to display alarm information.

#COUNT : (TYPE : DINT)

Is the total number of alarms defined. This number also sets how many registers are used for status registers, how many text screens are reserved for alarm display.

#MODE : (TYPE : DINT)

The mode in which the time stamping to be done.

STAMP[ ] : (TYPE : INT[])

The time & date values are stored in this array of registers.

## Remarks

### 1) The Control Block

**Alarm Count** - This is a BYTE counter that counts how many times an alarm occurs. The count only increments when the pending bit goes from low to high. To count another alarm event the alarm must be acknowledged, cleared and reactivated. When the count reaches a maximum of 255 it no longer changes until reset. This count can be reset by writing directly to this portion of the register using one of the BYTE instructions.

**Active** - This bit is set by the user's ladder program to indicate an alarm condition has occurred. For example, if the alarm is to indicate an over-temperature condition, have the ladder logic performs a compare, then set this bit if the compare indicates the temperature is greater than a set point.

**Pending** - This bit is set by the function block when the Active bit is high and is reset through the functions block's Clear operation. If the Active bit is high when Pending is reset, a new alarm will be recognized and Pending will be set immediately

**Acknowledge** - This bit is set by the function block after a pending alarm has been acknowledged.

#### Special Status Bits

Bit 16 of the first status word turns ON when any alarm is pending (but may be acknowledged).

Bit 15 of the first status word turn ON when any alarm is unacknowledged.

### 2) Time Stamp Registers

Time stamping can be set to one of three modes:

**None** - No time stamping is performed and no additional register space is required.

**Time Only** - The time is recorded when each alarm's pending bit becomes active. Each alarm requires three (3) registers starting at the block defined by the time stamping control block. The time is recorded in the same format as the real-time-clock is stored in the system registers.

**Time and Date** - The time and date is recorded when each alarm's pending bit becomes active. Each alarm requires six (6) registers starting at the block defined by the time stamping control block. The time and date is recorded in the same format as the real-time-clock is stored in the system registers.

#### ST Language

(\* ASMP is a declared instance of Alarm Stamp function block \*)

```
ASMP(CB[], NEXT, PREV, CLEAR, ACK, #SCREEN1, #COUNT, #MODE, STAMP[]);
```

#### FBD Language

#### LD Language

#### IL Language

(\* ASMP is a declared instance of Alarm Stamp function block \*)

```
OP1 : CAL ASMP(CB[], NEXT, PREV, CLEAR, ACK, #SCREEN1, #COUNT, #MODE, STAMP[])
```

See also

Alarm

## **ScaleInt**

*Operator* – This function scales the input to the specified range.

### **Inputs**

IN: The input which needs to be scaled to a specified range. (TYPE : INT)

MinIN: The minimum value of the input which has a defined scale. (TYPE : INT)

MaxIN: The maximum value of the input which has a defined scale. (TYPE : INT)

MinQ: The minimum value to which the input needs to be scaled. (TYPE : INT)

MaxQ: The maximum value to which the input needs to be scaled. (TYPE : INT)

### **Outputs**

Q : The scaled output in the range of MinQ & MaxQ specified. (TYPE : INT)

### **Remarks**

Cases often arise when numbers on one scale need to be translated to another scale. The MinIN and MaxIN ranges indicated the expected or nominal values that the input can be expected to attain. This is the range of values that corresponds to the expected output range.

The MinQ and MaxQ Ranges indicate the range of value that the input signal is converted to.

### **ST Language**

Q := ScaleInt(IN, MinIN, MaxIN, MinQ, MaxQ);

### **FBD Language**

### **LD Language**

### **IL Language**

```
Op1: LD IN
      ScaleInt MinIN, MaxIN, MinQ, MaxQ
      ST Q
```

### **See also**

[ScaleReal](#)

## **ScaleReal**

*Operator* – This function scales the input to the range specified.

### **Inputs**

IN: The input which needs to be scaled to a specified range. (TYPE : REAL)

MinIN: The minimum value of the input which has a defined scale. (TYPE : REAL)

MaxIN: The maximum value of the input which has a defined scale. (TYPE : REAL)

MinQ: The minimum value to which the input needs to be scaled. (TYPE : REAL)

MaxQ: The maximum value to which the input needs to be scaled. (TYPE : REAL)

### **Outputs**

Q : The scaled output in the range of MinQ & MaxQ specified. (TYPE : REAL)

### **Remarks**

Cases often arise when numbers on one scale need to be translated to another scale. The MinIN and MaxIN Ranges indicate the expected or nominal values that the input can be expected to attain.

The MinQ and MaxQ Ranges indicate the range of value that the input signal is converted to.

### **ST Language**

```
Q := ScaleReal(IN, MinIN, MaxIN, MinQ, MaxQ);
```

### **FBD Language**

### **LD Language**

### **IL Language**

```
Op1: LD IN  
      ScaleReal MinIN, MaxIN, MinQ, MaxQ  
      ST Q
```

### **See also**

[ScaleInt](#)



**SetClk**

*Operator* – Sets the clock, with the specified values.

**Inputs**

IN[ ] : Array of 6 elements, holds value in the form of SS MM HH, DD MM YYYY format. (TYPE : INT[])

**Outputs**

Q : This is set to TRUE if the operation of setting the clock is successful. (TYPE: BOOL)

**ST Language**

Q := SetClk(IN[]);

**FBD Language****LD Language****IL Language**

Not Available.

## STP100 Smart Stack Module

### Module Configuration

In operation, the Stepper Element writes all values as a group to the Stepper Controller SmartStack Module. (Technically, the actual write operation does not take place until the next **I/O cycle**.) This is a great convenience, as otherwise require six or ten individual elements. The Stepper Move instruction requires only one element.

The registers assigned to the Stepper Controller SmartStack Module are assigned by default when the controller is configured. The exact position of the module in the **%I**, **%Q**, **%AI**, and **%AQ** spaces is determined by the number of SmartStack modules attached to this controller, and the physical position of the HE800STP100 module within the stack.

This is a typical setup based on the HE800STP100 being the first (or only) SmartStack module, and indexing is not selected:

The image shows a 'Module Configuration' dialog box with two tabs: 'I/O Map' and 'Module Setup'. The 'Module Setup' tab is active. It displays the following information:

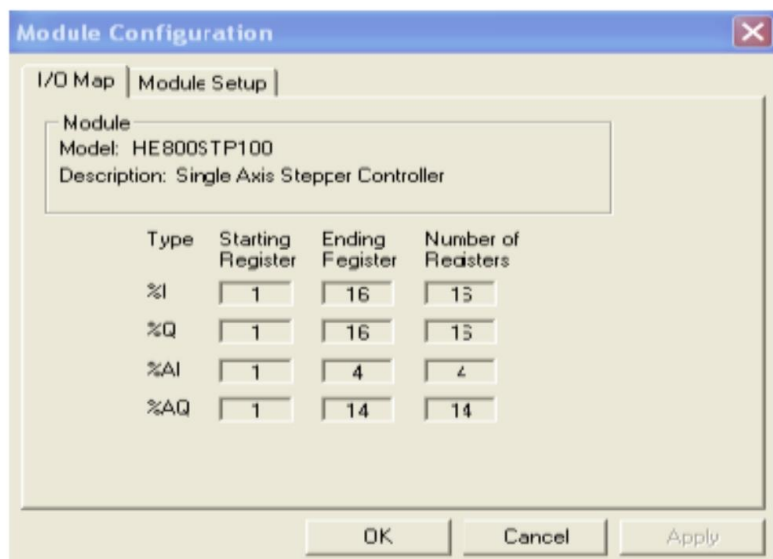
Module  
Model: HE800STP100  
Description: Single Axis Stepper Controller

Type	Starting Register	Ending Register	Number of Registers
%I	1	16	16
%Q	1	16	16
%AI	1	4	4
%AQ	1	7	7

At the bottom of the dialog are three buttons: 'OK', 'Cancel', and 'Apply'.

**Note:** The **STARTING LOCATION** indicated for this module, in particular those for **%AQ**. This information is used in the configuration screen. In this example, the Stepper Controller lives at address **%AQ01** and requires seven (7) consecutive registers. This information belongs in the **Stepper Starting %AQ** box of the element configuration screen.

**Note:** If the module *and the element* are configured to accept Indexed Moves, the element requires fourteen (14) consecutive **%AQ** registers as below.



First ensure that the SmartStack module is free to operate by checking the **Status Bits**, **%I1** to **%I16**. If *any* Error Bit is set, the source of the error must be cleared, and the **CLEAR ERRORS** command issued. Condition of the Status Bits depends on the previous command. Do not issue a new command (except the **IMMEDIATE STOP** or **DECELERATE AND STOP** command) until the previous command has completed.

When this element receives power, the values from the configured constants or registers are loaded into the HE800STP100, preparing it for the next command. (Technically, the actual write operation does not take place until the next **I/O cycle**.)

**Note: DO NOT** execute the Stepper Move element until the previous command is complete.

Commands are issued by setting the appropriate **command bit** in the Stepper Modules **% Q** address space after the Stepper Move element has completed.

#### COMMAND BITS

The sixteen- (16) Digital Output points (% Q) are used as Command Bits:

Point	Description
%Q1	Reserved
%Q2	Reserved
%Q3	Reserved
%Q4	Find Home Up
%Q5	Find Home Down

<b>%Q6</b>	Jog Up
<b>%Q7</b>	Jog Down
<b>%Q8</b>	Move Relative
<b>%Q9</b>	Move Absolute
<b>%Q10</b>	Resume Move
<b>%Q11</b>	Move Indexed
<b>%Q12</b>	Reserved
<b>%Q13</b>	Set Current Position
<b>%Q14</b>	Clear Errors
<b>%Q15</b>	Decelerate and Stop
<b>%Q16</b>	Immediate Stop

Only one command bit is active at one time. If more than one bit is active at one time, the bit with the highest number takes precedence. Note that this gives the IMMEDIATE STOP command the highest precedence.

Immediately after power up, the Power up Error Status Bit is TRUE. The CLEAR ERRORS command must be the first command issued. No other commands are accepted if any error bit is TRUE.

All command bits are positive (OFF to ON) edge sensitive. The JOG UP and JOG DOWN commands are also negative edge sensitive (ON to OFF) these commands require both begin and end signals.

**Note:** The CLEAR ERRORS command *must* be issued before any other command is issued. This is an important safety feature.

Not all commands are available at all times. For example, if a MOVE command is in progress, only the DECELERATE AND STOP or IMMEDIATE STOP commands are accepted.

#### **STATUS BITS**

The sixteen (16) Digital Input (%I) points are used as Status Bits:

<b>Point</b>	<b>Description</b>
<b>%I1</b>	Emergency Stop Error

<b>%I2</b>	Lower End Limit Stop Error
<b>%I3</b>	Upper End Limit Stop Error
<b>%I4</b>	Illegal Move Error
<b>%I5</b>	Motor Stalled Error
<b>%I6</b>	Future Use
<b>%I7</b>	Future Use
<b>%I8</b>	Power Up/Watch Dog Error
<b>%I9</b>	Position Valid
<b>%I10</b>	Current Position Valid
<b>%I11</b>	Future Use
<b>%I12</b>	Future Use
<b>%I13</b>	At Home
<b>%I14</b>	Accelerating
<b>%I15</b>	Decelerating
<b>%I16</b>	Moving

Bits 1 through 8 are Error Bits. The condition causing the error is present if the Error Bit is TRUE (1). The module does not function so long as any Error Bit is TRUE (1). These bits are cleared by issuing the CLEAR ERROR command.

Bit 8, Power Up/ WatchDog Error, is TRUE immediately after power up or watchdog timeout and prevents operation of the module until the CLEAR ERROR command is issued.

The CLEAR ERROR command must therefore be the first command issued. No other command is accepted while any error bit is TRUE.

Bits 9 through 16 are Status Bits. The status (TRUE or FALSE) of these bits indicates the status of the condition referenced by these bits. These are NOT errors, and the module continues to function normally in accordance with these bits. These bits are *not* affected by the CLEAR ERRORS command.

## POSITION FEEDBACK REGISTERS

The four- (4) Analog Input (%AI) points are used as two (2) DINT (32-bit) registers. The first two points are combined as a single 32-bit register, and the second two points are combined as a 32-bit register.

**Note:** Under *i<sup>3</sup> Configurator*, references to these register pairs would be specified as **DINT**.

Point	Description	Range
%AI1	Motor Position Low Word	-8,388,608 - +8,388,607
%AI2	Motor Position High Word	
%AI3	Encoder Position Low Word	-8,388,608 - +8,388,607
%AI4	Encoder Position High Word	

Immediately after reset, the value in these registers is 0 (zero) and is considered invalid as indicated by the CURRENT POSITION VALID Status Bit remaining FALSE.

The Motor Position value remains invalid until either FIND HOME command is issued or the SET CURRENT POSITION command is issued.

If the Motor Position is invalid, the MOVE ABSOLUTE command is not accepted.

## COMMAND DATA OUTPUTS

These registers contain the data by which the commands operate.

Point	Data Size	Description	Range
%AQ1	32-bit	Destination Low Word	-8,388,608 to +8,388,607
%AQ2		Destination High Word	
%AQ3	16-bit	Velocity Divisor	20 to 65,535
%AQ4	16-bit	Base Velocity	1. to 8,190
%AQ5	16-bit	Running Velocity	2. to 8,191
%AQ6	16-bit	Acceleration Time ( mS)	1. to 27,300
%AQ7	16-bit	Deceleration time ( mS)	0 to 27,300

The first two points are combined to form a single 32-bit register. This contains the location where the stepping stops. Depending on the instruction issued, this position is an absolute reference from the Home position or a relative position from the current position.

The **Velocity Divisor** determines the resolution for the Base Velocity and Running Velocity. Refer to the STP100 User Manual for a more complete discussion of this register.

The **Base Velocity** determines the first velocity used when a command starts, and the last velocity used when a command stops.

The **Running Velocity** is the top speed at which the command eventually operates.

In normal operation, a command starts operating at the Base Velocity, accelerates to the

**Running Velocity**, decelerates to the **Base Velocity**, then stops. The Accelerating, Decelerating, and Moving Status Bits reflects the current operational state.

The **Acceleration Time** is the amount of time the stepper allocates for accelerating between **Base Velocity** and **Running Velocity**.

**Deceleration Time** is the amount of time the stepper allocates for decelerating from **Running Velocity** to **Base Velocity**. If 0 (zero) is selected, the stepper automatically uses the **Acceleration Time** setting.

#### INDEXED MOVES

The STP100 performs indexed moves. To do so, the SmartStack module must be configured to accept an external Index Input, and the Stepper Move Element must also be configured to match.

**Note:** All Indexed Moves are relative.

Configuring the Stepper Move Element adds seven (7) additional registers, six of which are combined with each other to form three (3) 32-bit registers and one (1) 16-bit register.

Point	Data Size	Description	Range
%AQ8	32-bit	Indexed Destination Low Word	0 to 16,777,215
%AQ9		Indexed Destination High Word	
%AQ10	16-bit	Indexed Deceleration Time	0 to 27,300
%AQ11	32-bit	Index Window Begin Low Word	0 to 16,777,215
%AQ12		Index Window Begin High Word	

**%AQ13** <sup>32-bit</sup> Index Window End Low Word 0 to 16,777,215

**%AQ14** Index Window End High Word

The Index Move command looks at an external input called INDEX-. This normally expects to see a switch closure or some other electromechanical (optical, magnetic, etc.) device. The input is active LOW. If the Stepper Controller sees the INDEX- input low during the "window", the Stepper Controller moves the device to an alternate position.

The window is defined by the Index Window Begin Position and the Index Window End Position. The INDEX- input is honored *only* while the Stepper Controller is within this range.

**Note:** The window period is further limited to that time when the stepper has reached Running Velocity. If the window is defined such that the window attempts to open during acceleration, the window does not open until Running Velocity is reached. Also, the window closes automatically if the move starts to decelerate.

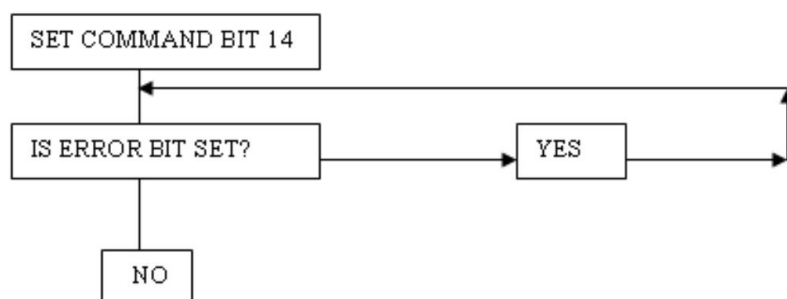
If the stepper never reaches Running Velocity the Index Window never opens.

If the INDEX- input occurs during the window, the Stepper Controller redefines the destination position of the move to be Indexed Destination Position, (%AQ8 / %AQ9) relative to the Current Motor Position (%AI1 / %AI2) at the time INDEX- became active. The deceleration of the move is determined by the Indexed Deceleration Time.

## ISSUING COMMANDS

The first step of issuing commands is to see that no errors exist. Immediately after Power Up or Reset, the Power Up Error Bit is set, so the first command issued must be the CLEAR ERRORS command.

A simple flow chart indicates how the CLEAR ERRORS command is affected:



After the Power On Error is cleared, the Current Position is not valid. This is noted by Status Bit 10 being FALSE (0). The program needs to issue a FIND HOME UP,



FIND HOME DOWN, or SET CURRENT POSITION command in order to validate the position.

Current Position can become invalid (0) if the motor stops suddenly. This can be caused by an Emergency Stop, Lower Limit Error, Upper Limit Error, Motor Stalled Error or by issuing an IMMEDIATE STOP command. In any case, any source of error must be corrected, and the motor homed or the current position updated as outlined.

Other commands are issued in a similar manner:

1. If there are any errors present, correct the source of the errors then issue the Clear Errors command.
2. Setup the values for the Stepper Move element, and then apply power to the Stepper Move element.
3. Set the appropriate Command Bit to 1
4. Check the appropriate status bits for the command.
5. Do not issue another command until this command either completes successfully or errors out.

## **StepperMove**

*Operator* -The Stepper Move element provides the necessary interface between *i<sup>3</sup> Configurator* and the IMO STP100 Single Axis Stepper Controller SmartStack module with no index.

### **Inputs**

STEPPER STARTING % AQ (@Stepper): (TYPE: INT)

This contains the address of the first % AQ register assigned to the Stepper SmartStack module. This information can be taken from the Stepper Module SmartStack Configuration.

DESTINATION POSITION ( DP): (TYPE: DINT)

This is a 32-bit register, contains the position where the move is to end. Value range is -8,388,608 to +8,388,607

VELOCITY RESOLUTION ( VR): (TYPE: INT)

This is a 16-bit register. Values range from 20 to 65535.

BASE VELOCITY ( BV): (TYPE: INT)

This is a 16-bit register. Values range from 1 to 8190.

RUNNING VELOCITY ( RV): (TYPE: INT)

This is a 16-bit register. Values range from 2 to 8191.

ACCELERATION TIME (AC): (TYPE: INT)

This is a 16-bit register. Times are listed in milliseconds ( mS). Values range from 1 to 27300.

DECELERATION TIME (DC): (TYPE: INT)

This is a 16-bit register. Times are listed in milliseconds ( mS). Values range from 0 to 27300.

### **Remarks**

Configuration

NOTE: Verify the SmartStack module configuration before completing the Element Configuration. The various entries must be completed by the programmer. Values can be entered as numeric constants or registers reference by Name.

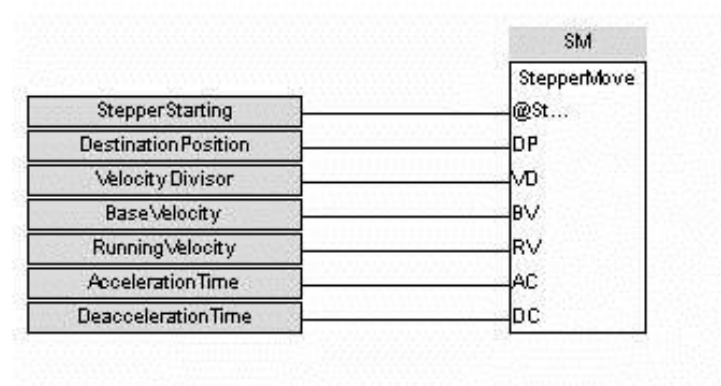
The STP100 module requires either seven (7) or fourteen (14) consecutive Analog Output (% AQ) registers. To program the STP100 module, appropriate data must be moved into the assigned % AQ registers, typically using seven or 14 Move Word elements.

The purpose of the Stepper Move element is to transfer this data to the STP100 module with one instruction. Additionally, the Stepper Move element provides a built in Stepper Motion Calculator that can calculate a Movement Profile graph based on user-selected values.

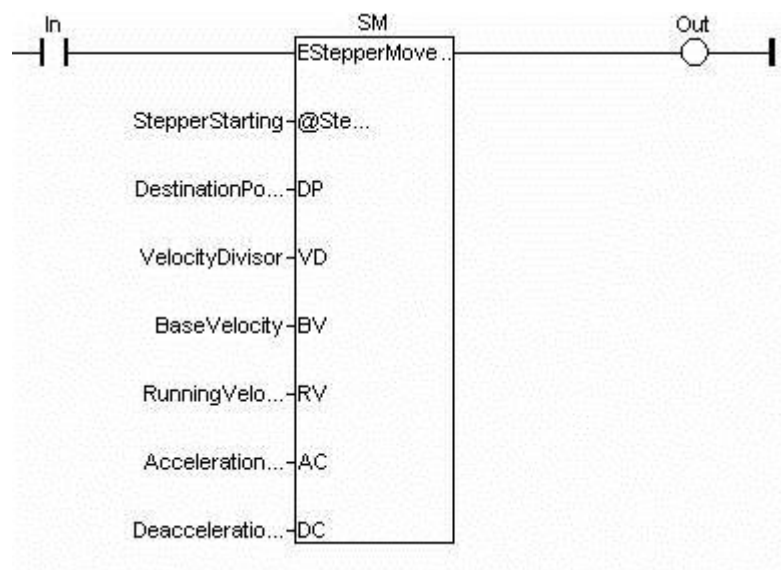
### ST Language

(\* STEPМ is a declared instance of StepperMove function block \*)  
STEPМ(@Stepper, DP, VD, BV, RV, AC, DC);

### FBD Language



### Ladder Language



### IL Language

(\* STEPМ is a declared instance of StepperMove function block \*)  
Op1: CAL STEPМ(@Stepper, DP, VD, BV, RV, AC, DC)

### See also

StepperMoveInd STP100

## **StepperMoveInd**

*Operator* - Stepper Move element provides the necessary interface between *i<sup>3</sup> Configurator* and IMO's STP100 Single Axis Stepper Controller SmartStack module with Index defined.

### Inputs

STEPPER STARTING % AQ (@Stepper): (TYPE: INT)

This contains the address of the first % AQ register assigned to the Stepper SmartStack module. This information can be taken from the Stepper Module SmartStack Configuration.

DESTINATION POSITION ( DP): (TYPE: DINT)

This is a 32-bit register, contains the position where the move is to end. Value range is -8,388,608 to +8,388,607

VELOCITY RESOLUTION ( VR): (TYPE: INT)

This is a 16-bit register. Values range from 20 to 65535.

BASE VELOCITY ( BV): (TYPE: INT)

This is a 16-bit register. Values range from 1 to 8190.

RUNNING VELOCITY ( RV): (TYPE: INT)

This is a 16-bit register. Values range from 2 to 8191.

ACCELERATION TIME (AC): (TYPE: INT)

This is a 16-bit register. Times are listed in milliseconds ( mS). Values range from 1 to 27300.

DECELERATION TIME (DC): (TYPE: INT)

This is a 16-bit register. Times are listed in milliseconds ( mS). Values range from 0 to 27300.

### **The following registers are used only for Index Move operations:**

INDEX DESTINATION POSITION ( IDP): (TYPE: DINT)

This is a 32-bit register. Values range from 0 to 16,777,215.

INDEX DECELERATION ( IDC): (TYPE: INT)

This is a 16-bit register. Values range from 1 to 27,300.

INDEX WINDOW OPEN (IWO): (TYPE: DINT)

This is a 32-bit register. Values range from 0 to 16,777,215.

INDEX WINDOW CLOSED ( IWC): (TYPE: DINT)

This is a 32-bit register. Values range from 0 to 16,777,215.

### Remarks

Configuration

NOTE: Verify the SmartStack module configuration before completing the Element Configuration. The various entries must be completed by the programmer. Values can be entered as numeric constants or registers reference by Name.

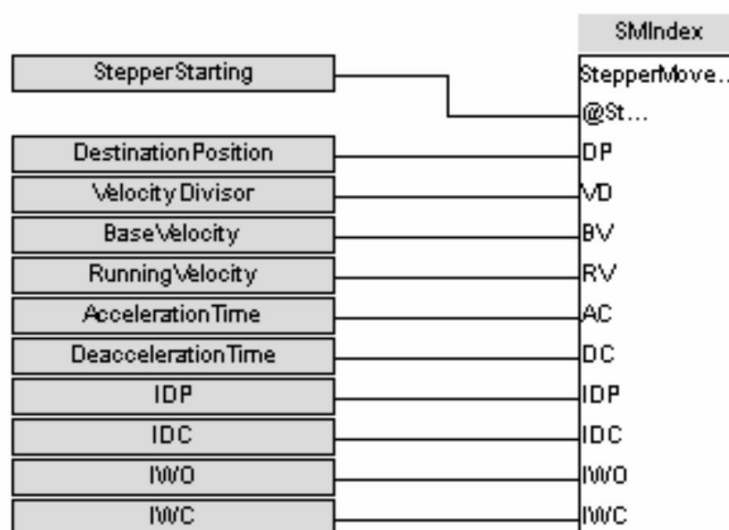
The STP100 module requires either seven (7) or fourteen (14) consecutive Analog Output (% AQ) registers. To program the STP100 module, appropriate data must be moved into the assigned % AQ registers, typically using seven or 14 Move Word elements.

The purpose of the Stepper Move element is to transfer this data to the STP100 module with one instruction. Additionally, the Stepper Move element provides a built in Stepper Motion Calculator that can calculate a Movement Profile graph based on user-selected values.

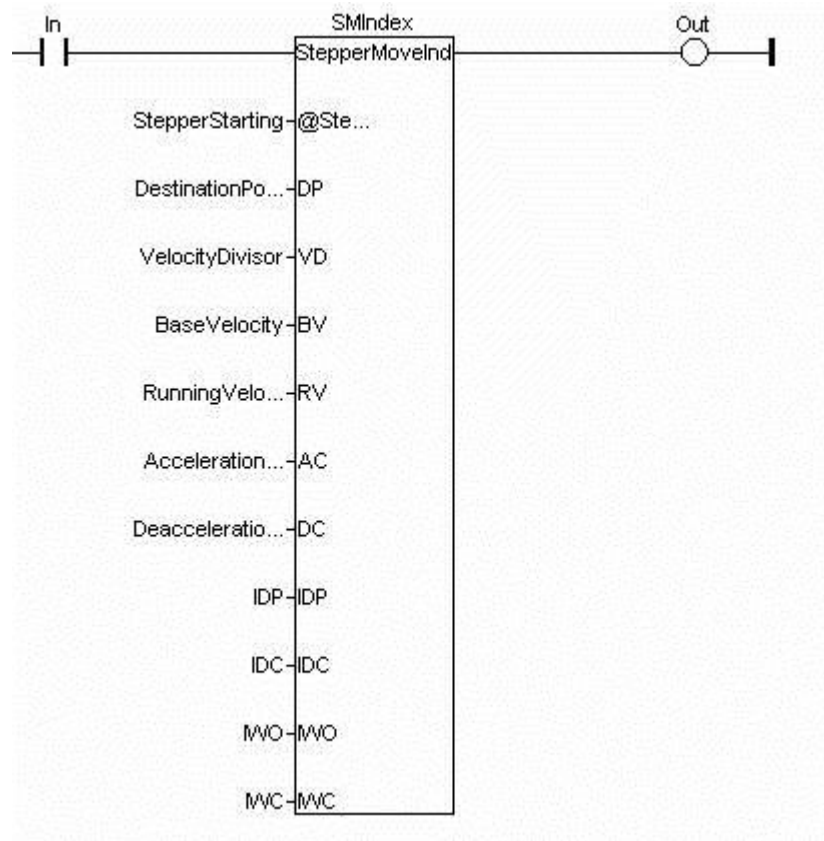
### ST Language

(\* STEPMP is a declared instance of StepperMoveInd function block \*)  
 STEPMP (@Stepper, DP, VD, BV, RV, AC, DC, IDP, IDC, IWO, IWC);

### FBD Language



### Ladder Language



#### IL Language

(\* STEPMI is a declared instance of StepperMoveInd function block \*)

Op1 : CAL STEPMI (@Stepper, DP, VD, BV, RV, AC, DC, IDP, IDC, IWO, IWC)

#### See also

StepperMove STP100

## Key Press

*Operator* - PressKey block provides a facility to add replicate key to existing Function keys, soft keys and other front panel keys of the *i3* with exception to system key of Touch-*i3*s.

**Note:-** Only Digital input bits can be set as Replicate Keys.

### Inputs

**#Key:** Key Number (TYPE : DINT)

**@ReplKey:** Key to replicate (TYPE: BOOL)

### Outputs

**Q :** Output (Type: BOOL)

### Remarks

The operation of the Replicate Keys (PressKey) is exactly same as the actual keys operations but in *i3* run mode only. The Replicate Key works in parallel to actual Keys.

### ST Language

```
-  
Q:= KEYPRESS(16#1, Input1(*BOOL*) );  
Q1:=F1_Key;  
Q11:= Input1;
```

### FBD Language

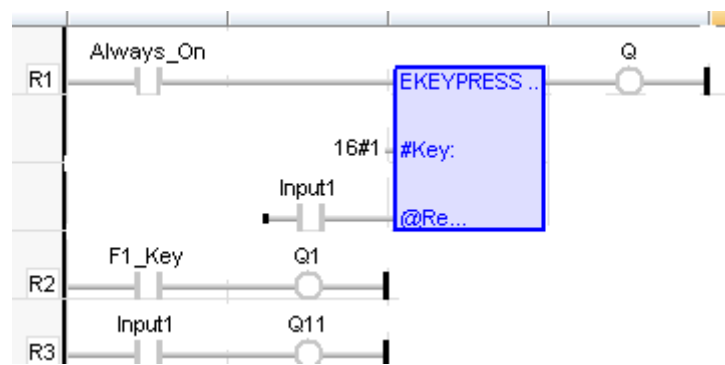
-



-

### LD Language

-



### IL Language

BEGIN\_IL

```
LD 1
KEYPRESS Input1
S Q1
```

END\_IL



## LoadRcpByIndex

*Operator* - The load function allows an operator to select a product from the specified recipe. Once selected the settings for the appropriate recipe will be loaded in to the registers specified for the recipe.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

#Rcp - (Type: DINT)

Recipe number. First Recipe = 0, Second Recipe = 1 and so on.

Index - (Type: DINT)

Index specifies the Product. Here First Product = 1, Second Product = 2 and so on.

@Status - (Type: DINT)

A 16-bit register used to hold the results of the element.

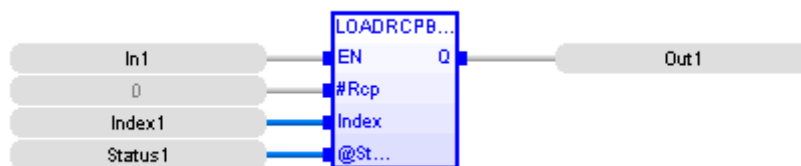
### Outputs

**Q** : Output (Type: BOOL)

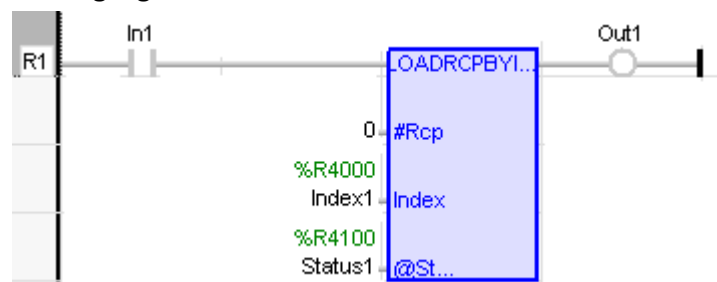
ST Language

Q := LOADRCPBYINDEX( EN1(\*BOOL\*), 16#0 (\*DINT\*), 1(\*DINT\*),  
Status(\*DINT\*) );

### FBD Language



### LD Language



### IL Language

BEGIN\_IL

LD LRBI\_IN1

LOADRCPBYINDEX 0, 1, Status1

ST LRBI\_Out1

END\_IL

## LoadRcpByStr

*Operator* - The load function allows an operator to select a product from the specified recipe. Once selected the settings for the appropriate recipe will be loaded in to the registers specified for the recipe.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

#Rcp - (Type: DINT)

Recipe number. First Recipe = 0, Second Recipe = 1 etc.

#Name - (Type: String)

Name of the Product

@Status - (Type: DINT)

A 16-bit register used to hold the results of the element.

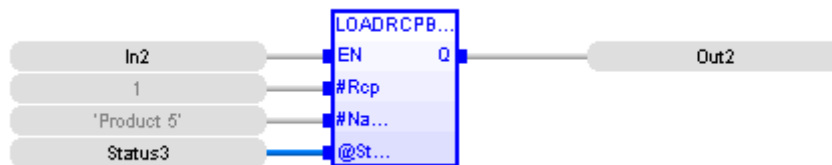
Outputs

Q : Output (Type: BOOL)

### ST Language

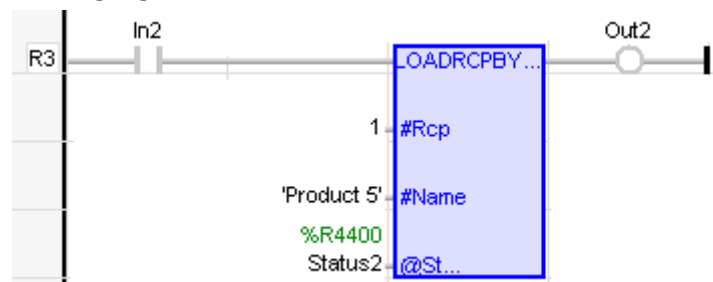
```
Q1:= LOADRCPBYSTR( EN2(*BOOL*), 16#1(*DINT*), 'Product 5'(*STRING*),  
Status3(*DINT*) );
```

### FBD Language



-

### LD Language



-

### IL Language

```
BEGIN_IL
```

```
LD LRBS_IN2
```

```
LOADRCPBYSTR 1, 'Product 5', Status2
```

```
ST LRBS_Q1
```

END\_IL

## LoadRcpByStr2

*Operator* - The load function allows an operator to select a product from the specified recipe. Once selected the settings for the appropriate recipe will be loaded in to the registers specified for the recipe.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

#Rcp - (Type: DINT)

Recipe number. First Recipe = 0, Second Recipe = 1 etc.

Name[] - (Type: USINT[])

Name of the Product assigned through array of registers.

@Status - (Type: DINT)

A 16-bit register used to hold the results of the element.

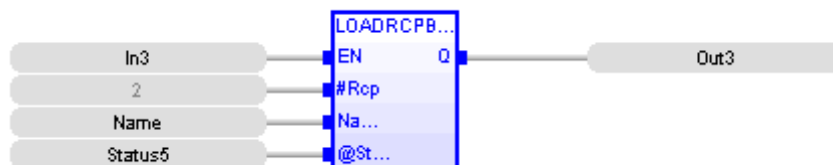
### Outputs

Q : Output (Type: BOOL)

### ST Language

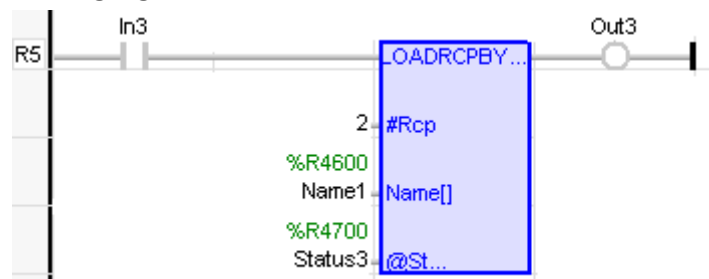
```
Q2 := LOADRCPBYSTR2( EN3(*BOOL*), 16#2(*DINT*), Name(*USINT*),  
Status5(*DINT*) );
```

### FBD Language



-

### LD Language



-

### IL Language

```
BEGIN_IL
```

LD LRBS2\_IN1  
LOADRCPBYSTR2 2, Name1, Status3  
ST LRBS2\_Out3

END\_IL

## SaveRcpByIndex

*Operator* - The Save function allows an operator to save the current working settings for a recipe to a product setting. For example if the settings for a previously saved product have been altered for improved machine operation, the operator can save the adjusted settings back to the recipe database.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

#Rcp - (Type: DINT)

Recipe number. First Recipe = 0, Second Recipe = 1 and so on.

Index - (Type: DINT)

Index specifies the Product. Here First Product = 1, Second Product = 2 and so on.

@Status - (Type: DINT)

A 16-bit register used to hold the results of the element.

### Outputs

Q : Output (Type: BOOL)

### ST Language

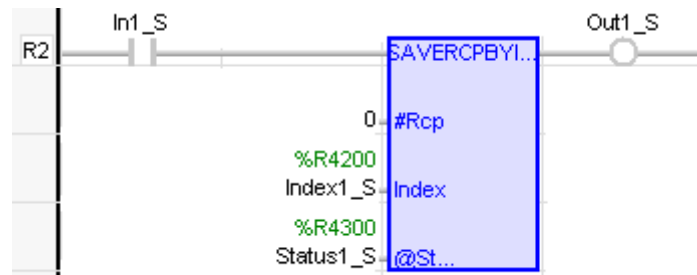
```
Q:=SAVERCPBYINDEX( EN1(*BOOL*), 0(*DINT*), 2(*DINT*), Status2(*DINT*) );
```

### FBD Language



-

### LD Language



-

### IL Language

```
BEGIN_IL
```

```
LD SRBI_IN1_S
```

SAVERCPBYINDEX 0, 2, Status2  
ST SRBI\_Out1\_S

END\_IL



## SaveRcpByStr

*Operator* - The Save function allows an operator to save the current working settings for a recipe to a product setting. For example if the settings for a previously saved product have been altered for improved machine operation, the operator can save the adjusted settings back to the recipe database.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

#Rcp - (Type: DINT)

Recipe number. First Recipe = 0, Second Recipe = 1 etc.

#Name - (Type: String)

Name of the Product

@Status - (Type: DINT)

A 16-bit register used to hold the results of the element.

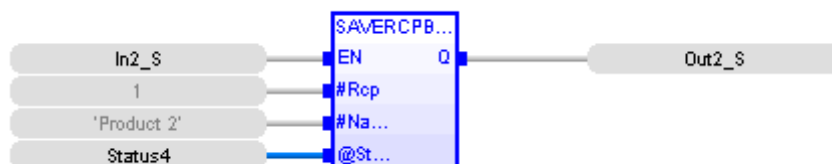
### Outputs

Q : Output (Type: BOOL)

### ST Language

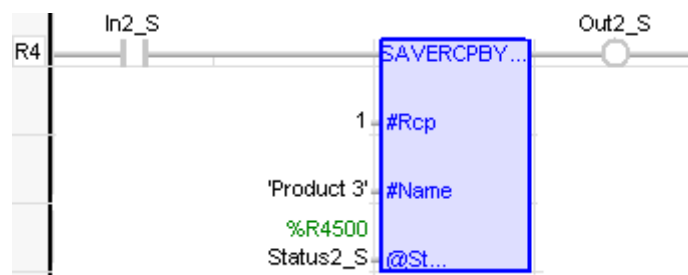
```
Q1:=SAVERCPBYSTR( EN2(*BOOL*), 1(*DINT*), 'Product 3'(*STRING*),  
Status2_S(*DINT*) );
```

### FBD Language



-

### LD Language



-

### IL Language

```
BEGIN_IL
```

LD SRBS\_IN2\_S  
SAVERCPBYSTR 1, 'Product 3', Status2\_S  
ST SRBS\_Out\_S

END\_IL

## SaveRcpByStr2

*Operator* - The Save function allows an operator to save the current working settings for a recipe to a product setting. For example if the settings for a previously saved product have been altered for improved machine operation, the operator can save the adjusted settings back to the recipe database.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

#Rcp - (Type: DINT)

Recipe number. First Recipe = 0, Second Recipe = 1 etc.

Name[] - (Type: USINT[])

Name of the Product assigned through array of registers.

@Status - (Type: DINT)

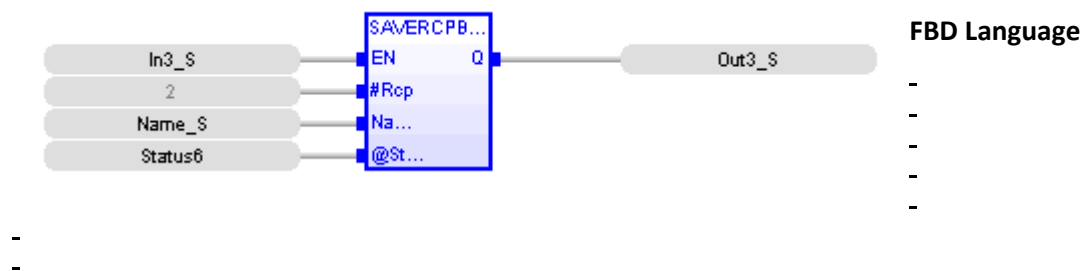
A 16-bit register used to hold the results of the element.

### Outputs

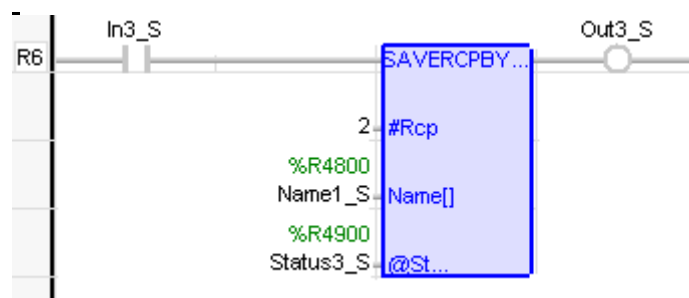
Q : Output (Type: BOOL)

### ST Language

```
Q2:=SAVERCPBYSTR2( EN3(*BOOL*), 2(*DINT*), Name_S(*USINT*), Status6(*DINT*) );
```



### LD Language



### IL Language

```
BEGIN_IL
```

```
LD SRBS2_IN3_S
```

```
SAVERCPBYSTR2 0, Name1_S, Status3_S
```

```
ST SRBS2_Out3_S
```

END\_IL

## Register Operations

### *Register Operations*

Below are the standard functions for managing registers:

Shift Left	shift left
Shift Right	shift right
Rotate Left	rotation left
Rotate Right	rotation right
BitSet	Bit Set
BitClear	Bit Clear
BitTest	Bit Test

The following functions enable bit to bit operations on an registers:

AND	boolean AND
OR	boolean OR
XOR	exclusive OR
NOT	boolean negation

## AND\_MASK

*Function* - Performs a bit to bit AND between two register values

### Inputs

IN : ANY First input

MSK : ANY Second input (AND mask)

### Outputs

Q : ANY AND mask between IN and MSK inputs

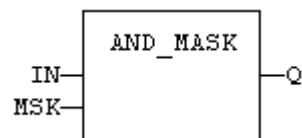
### Remarks

In LD language, the EN signal enables the operation, and the ENO keeps the same value as the EN. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

### ST Language

Q := AND\_MASK (IN, MSK);

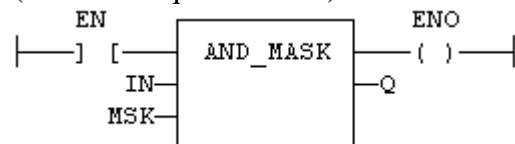
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### IL Language

Op1: LD IN

AND\_MASK MSK

ST Q

### See also

OR\_MASK XOR\_MASK NOT\_MASK

## NOT\_MASK

*Function* - Performs a bit to bit negation of an register value

### Inputs

IN : ANY register input

### Outputs

Q : ANY Bit to bit negation of the input

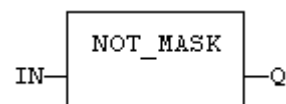
### Remarks

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the parameter (IN) must be loaded in the current result before calling the function.

### ST Language

Q := NOT\_MASK (IN);

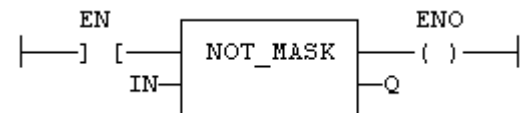
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### IL Language

```
Op1: LD    IN
      NOT_MASK
      ST    Q
```

### See also

AND\_MASK OR\_MASK XOR\_MASK

## OR\_MASK

*Function* - Performs a bit to bit OR between two register values

### Inputs

IN : ANY First input

MSK : ANY Second input (OR mask)

### Outputs

Q : ANY OR mask between IN and MSK inputs

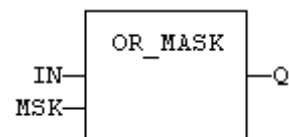
### Remarks

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

### ST Language

Q := OR\_MASK (IN, MSK);

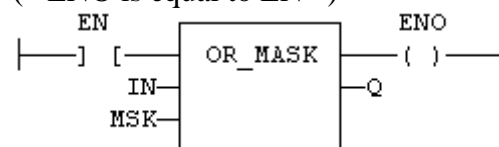
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### IL Language

```
Op1: LD    IN
      OR_MASK MSK
      ST    Q
```

### See also

AND\_MASK XOR\_MASK NOT\_MASK



## ROL

*Function* - Rotate bits of a register to the left.

### Inputs

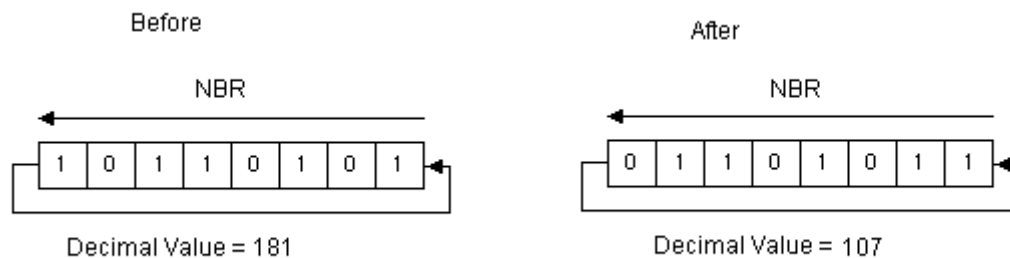
IN : ANY register

N : ANY Number of rotations (each rotation is 1 bit)

### Outputs

Q : ANY Rotated register

### Diagram



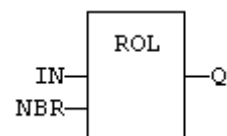
### Remarks

In LD language, the EN signal enables the operation, and the ENO keeps the state of the EN. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

### ST Language

Q := ROL (IN, NBR);

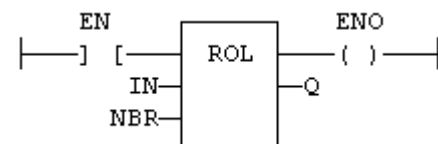
### FBD Language



### LD Language

(\* The rotation is executed only if EN is TRUE \*)

(\* ENO has the same value as EN \*)



### IL Language

```
Op1: LD IN
      ROL NBR
      ST Q
```

***See also***

SHL SHR ROR BitSet BitClear BitTest

## ROR

*Function* - Rotate bits of a register to the right.

### Inputs

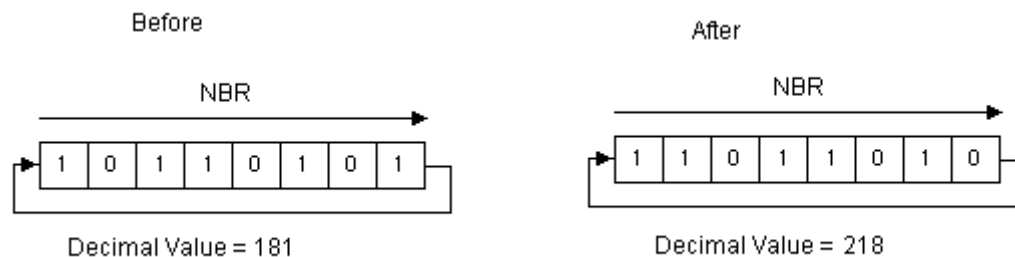
IN : ANY register

N : ANY Number of rotations (each rotation is 1 bit)

### Outputs

Q : ANY Rotated register

### Diagram



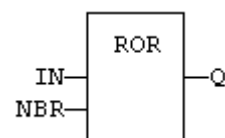
### Remarks

In LD language, the EN signal enables the operation, and the ENO keeps the state of the EN. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

### ST Language

Q := ROR (IN, NBR);

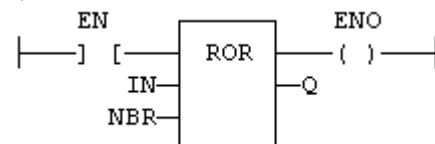
### FBD Language



### LD Language

(\* The rotation is executed only if EN is TRUE \*)

(\* ENO has the same value as EN \*)



### IL Language

```
Op1: LD IN
      ROR NBR
      ST Q
```

See also

SHL SHR ROL BitSet BitClear BitTest

## SHL

*Function* - Shift bits of a register to the left.

### Inputs

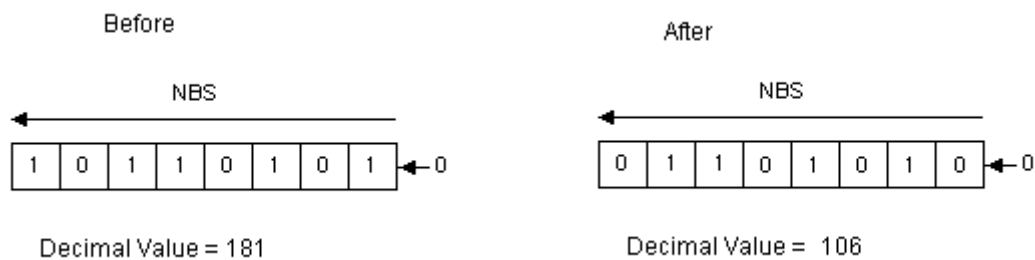
IN : ANY register

N : ANY Number of shifts (each shift is 1 bit)

### Outputs

Q : ANY Shifted register

### Diagram



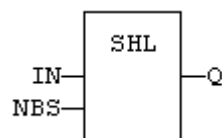
### Remarks

In LD language, the EN signal enables the operation, and the ENO keeps the state of the EN. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

### ST Language

Q := SHL (IN, NBS);

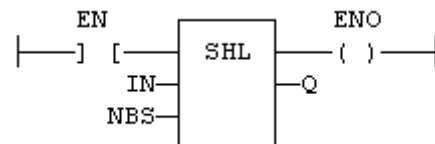
### FBD Language



### LD Language

(\* The shift is executed only if EN is TRUE \*)

(\* ENO has the same value as EN \*)



### IL Language

Op1: LD IN  
SHL NBS  
ST Q

### See also

SHR ROL ROR BitSet BitClear BitTest

## SHR

*Function* - Shift bits of a register to the right.

### Inputs

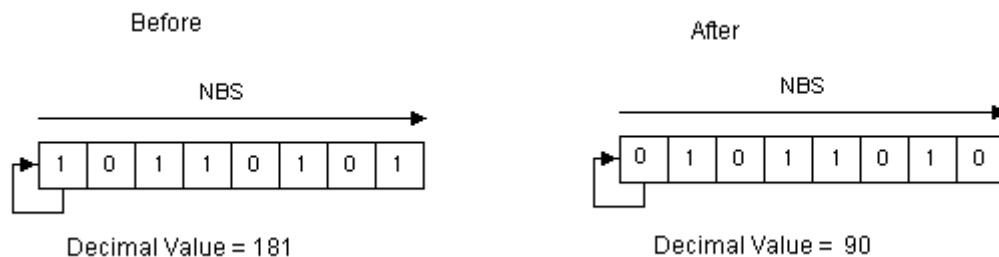
IN : ANY register

N : ANY Number of shifts (each shift is 1 bit)

### Outputs

Q : ANY Shifted register

### Diagram



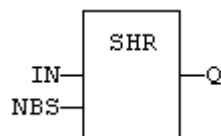
### Remarks

In LD language, the EN signal enables the operation, and the ENO keeps the state of the EN. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

### ST Language

Q := SHR (IN, NBS);

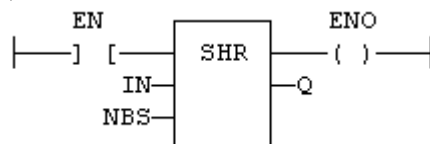
### FBD Language



### LD Language

(\* The shift is executed only if EN is TRUE \*)

(\* ENO has the same value as EN \*)



### IL Language

```
Op1: LD IN
      SHR NBS
      ST Q
```

### See also

SHL ROL ROR BitSet BitClear BitTest



## XOR\_MASK

*Function* - Performs a bit to bit exclusive OR between two register values

### Inputs

IN : ANY First input

MSK : ANY Second input (XOR mask)

### Outputs

Q : ANY Exclusive OR mask between IN and MSK inputs

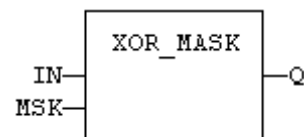
### Remarks

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

### ST Language

Q := XOR\_MASK (IN, MSK);

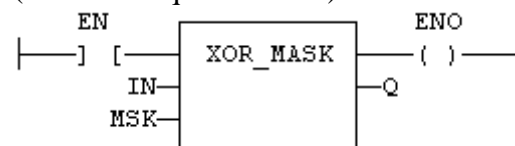
### FBD Language



### LD Language

(\* The function is executed only if EN is TRUE \*)

(\* ENO is equal to EN \*)



### IL Language

```
Op1: LD    IN
      XOR_MASK MSK
      ST    Q
```

### See also

AND\_MASK OR\_MASK NOT\_MASK

## BitSet

**Note:** Supported by Firmware 12.70 or above.

### Inputs

IN: Any data type, typically an array.

LEN: The size of the block in bytes.

BIT: The number of the bit in the block to be set.

### Outputs

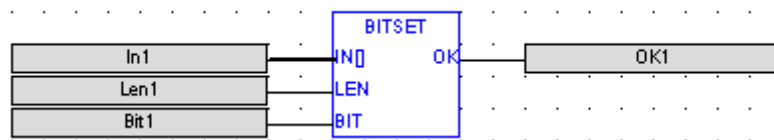
Q : Bit valid. TRUE if the bit number is  $\leq \text{LEN} \times 8$

This element sets a bit in a bit string (IN) to 1. Bit string length (LEN) can be between 1 to 512 bytes.

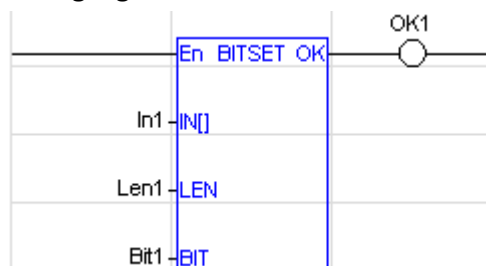
### ST Language

OK1 := BITSET( In1, Len1, Bit1);

### FBD Language



### LD Language



### IL Language

Not Available

### See also

SHL SHR ROL ROR BitClear BitTest

## BitTest

**Note:** Supported by Firmware 12.70 or above.

### Inputs

IN: Any data type, typically an array.

LEN: The size of the block in bytes.

BIT: The number of the bit in the block to be tested.

### Outputs

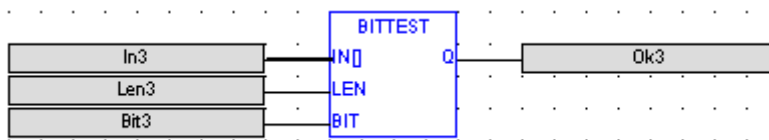
Q : Bit valid. TRUE if the bit number is  $\leq \text{LEN} \times 8$  AND the bit specified is TRUE.

This element tests a bit in a bit string (IN) to 1. Bit string length (LEN) can be between 1 to 512 bytes.

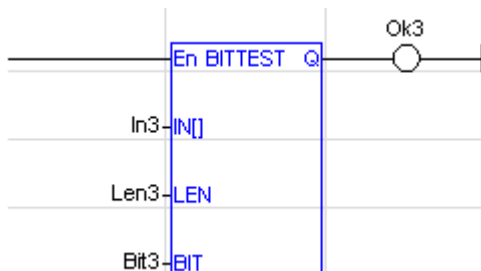
### ST Language

OK3 := BITTEST( In3, Len3, Bit3);

### FBD Language



### LD Language



### IL Language

Not Available

### See also

SHL SHR ROL ROR BitSet BitClear

## BitClear

**Note:** Supported by Firmware 12.70 or above.

### Inputs

IN: Any data type, typically an array.

LEN: The size of the block in bytes.

BIT: The number of the bit in the block to be cleared.

### Outputs

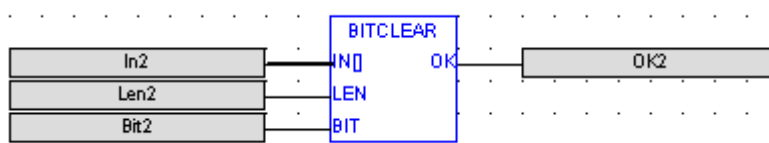
Q : Bit valid. TRUE if the bit number is  $\leq \text{LEN} \times 8$

This element clears a bit in a bit string (INe) to 1. Bit string length (LEN) can be between 1 and 512 bytes.

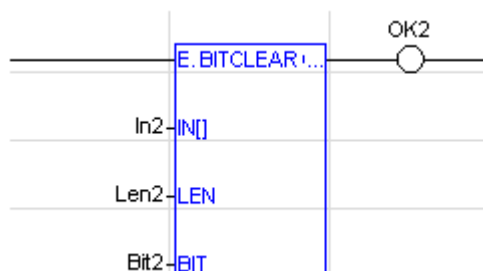
### ST Language

OK2 := BITCLEAR( In2, Len2, Bit2);

### FBD Language



### LD Language



### IL Language

Not Available

### See also

SHL SHR ROL ROR BitSet BitTest

## Conversion Operations

### *Conversion Operations*

Below are the standard functions for converting a data into another data type:

Convert to Boolean	converts to boolean
Convert to Small Integer	converts to small (8 bit) integer
Convert to INT16	converts to 16 bit integer
Convert to INT32	converts to integer (32 bit - default)
Convert to Real	converts to real
Convert to Time	converts to time
Any_To_String	converts to character string

AnyToBool

*Operator* - Converts the input into boolean value.

### **Inputs**

IN : ANYInput value

### **Outputs**

Q : BOOLValue converted to boolean

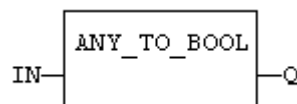
### **Remarks**

For DINT, REAL and TIME input data types, the result is FALSE if the input is 0. The result is TRUE in all other cases. For STRING inputs, the output is TRUE if the input string is not empty, and FALSE if the string is empty. In LD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung is the result of the conversion. In IL Language, the ANY\_TO\_BOOL function converts the current result.

### **ST Language**

Q := ANY\_TO\_BOOL (IN);

### **FBD Language**

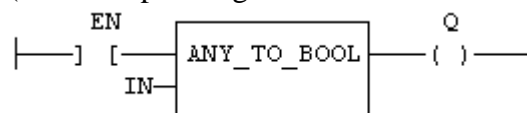


### **LD Language**

(\* The conversion is executed only if EN is TRUE \*)

(\* The output rung is the result of the conversion \*)

(\* The output rung is FALSE if the EN is FALSE \*)



### **IL Language**

Op1: LD IN

ANY\_TO\_BOOL

ST Q

### **See also**

ANY\_TO\_SINT ANY\_TO\_INT ANY\_TO\_DINT ANY\_TO\_REAL ANY\_TO\_  
TIME ANY\_TO\_STRING

ANY\_TO\_INT / ANY\_TO\_UINT

*Operator* - Converts the input into 16 bit integer value.

### **Inputs**

IN : ANY Input value

### **Outputs**

Q : INT Value converted to 16 bit integer

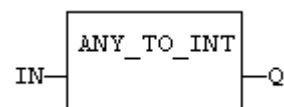
### **Remarks**

For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL Language, the ANY\_TO\_INT function converts the current result.

### **ST Language**

Q := ANY\_TO\_INT (IN);

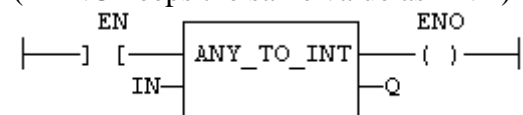
### **FBD Language**



### **LD Language**

(\* The conversion is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

Op1: LD IN

ANY\_TO\_INT

ST Q

### **See also**

ANY\_TO\_BOOL ANY\_TO\_SINT ANY\_TO\_DINT ANY\_TO\_REAL ANY\_T  
O\_TIME ANY\_TO\_STRING

AnyToString

*Operator* - Converts the input into string value.

### **Inputs**

IN : ANY Input value

### **Outputs**

Q : STRINGValue converted to string

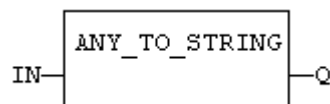
### **Remarks**

For BOOL input data types, the output is '1' or '0' for TRUE and FALSE respectively. For DINT, REAL or TIME input data types, the output is the string representation of the input number. This is a number of milliseconds for TIME inputs. In LD language, the conversion is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL language, the ANY\_TO\_STRING function converts the current result.

### **ST Language**

Q := ANY\_TO\_STRING (IN);

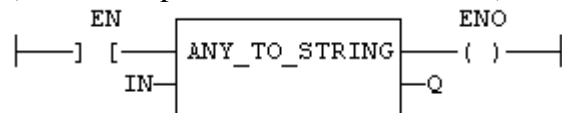
### **FBD Language**



### **LD Language**

(\* The conversion is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

Op1: LD IN

ANY\_TO\_STRING

ST Q

### **See also**

ANY\_TO\_BOOL ANY\_TO\_SINT ANY\_TO\_INT ANY\_TO\_DINT ANY\_TO\_REAL ANY\_TO\_TIME



AnyToSint / AnyToUsint

*Operator* - Converts the input into a small (8 bit) integer value.

### **Inputs**

IN : ANY Input value

### **Outputs**

Q : SINT Value converted to a small (8 bit) integer

### **Remarks**

For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL Language, the ANY\_TO\_SINT function converts the current result.

### **ST Language**

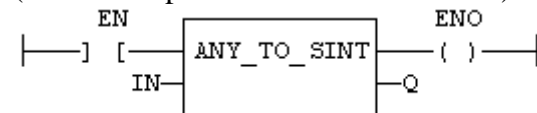
Q := ANY\_TO\_SINT (IN);

### **FBD Language**

### **LD Language**

(\* The conversion is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

Op1: LD IN

ANY\_TO\_SINT

ST Q

### **See also**

ANY\_TO\_BOOL ANY\_TO\_INT ANY\_TO\_DINT ANY\_TO\_REAL ANY\_TO\_TIME ANY\_TO\_STRING

AnyToTime

*Operator* - Converts the input into time value.

### **Inputs**

IN : ANY Input value

### **Outputs**

Q : TIME Value converted to time

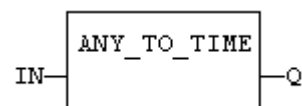
### **Remarks**

For BOOL input data types, the output is t#0ms or t#1ms. For DINT or REAL input data type, the output is the time represented by the input number as a number of milliseconds. For STRING inputs, the output is the time represented by the string, or t#0ms if the string does not represent a valid time. In LD language, the conversion is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL Language, the ANY\_TO\_TIME function converts the current result.

### **ST Language**

Q := ANY\_TO\_TIME (IN);

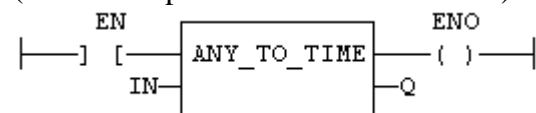
### **FBD Language**



### **LD Language**

(\* The conversion is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

Op1: LD IN

ANY\_TO\_TIME

ST Q

### **See also**

ANY\_TO\_BOOL ANY\_TO\_SINT ANY\_TO\_INT ANY\_TO\_DINT ANY\_T  
O\_REAL ANY\_TO\_STRING

ANY\_TO\_DINT / ANY\_TO\_UDINT

*Operator* - Converts the input into integer value.

### **Inputs**

IN : ANY Input value

### **Outputs**

Q : DINT Value converted to integer

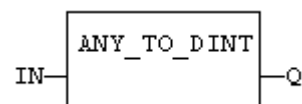
### **Remarks**

For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL Language, the ANY\_TO\_DINT function converts the current result.

### **ST Language**

Q := ANY\_TO\_DINT (IN);

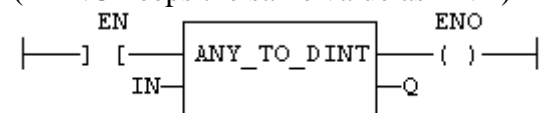
### **FBD Language**



### **LD Language**

(\* The conversion is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

Op1: LD IN

ANY\_TO\_DINT

ST Q

### **See also**

ANY\_TO\_BOOL ANY\_TO\_SINT ANY\_TO\_INT ANY\_TO\_REAL ANY\_TO\_TIME ANY\_TO\_STRING

AnyToReal

*Operator* - Converts the input into real value.

### **Inputs**

IN : ANY Input value

### **Outputs**

Q : REAL Value converted to real

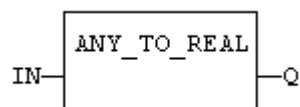
### **Remarks**

For BOOL input data types, the output is 0.0 or 1.0. For DINT input data type, the output is the same number. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number. In LD language, the conversion is executed only if the EN is TRUE. The ENO keeps the same value as the EN. In IL Language, the ANY\_TO\_REAL function converts the current result.

### **ST Language**

Q := ANY\_TO\_REAL (IN);

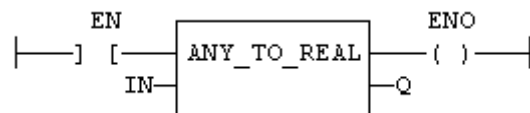
### **FBD Language**



### **LD Language**

(\* The conversion is executed only if EN is TRUE \*)

(\* ENO keeps the same value as EN \*)



### **IL Language**

Op1: LD IN

ANY\_TO\_REAL

ST Q

### **See also**

ANY\_TO\_BOOL ANY\_TO\_SINT ANY\_TO\_INT ANY\_TO\_DINT ANY\_TO\_TIME ANY\_TO\_STRING

## String Operations

### *String Operations*

Below are the standard operators and functions that manage character strings:

Set String	Initialise an array with a constant string
String Length	Calculate a string length
CmpStringConst	Compare a string with a constant
CmpStringVar	Compare two strings
ASCII To DINT	Convert a string to a 32bit word
ASCII To INT	Convert a string to an 16bit word
ASCII To Real	Convert a string to a floating point value
DINT To ASCII	Convert a 32bit word to a string
INT To ASCII	Convert a 16bit word to a string
Real To ASCII	Convert a floating point value to a string

**AsciiToDint**

*Operator* – Perform conversion of ASCII to a Dint value.

**Inputs**

SRC[ ] : (TYPE :USINT[])

SRC is the array of ASCII value placed at the input.

**Outputs**

Q : (TYPE :DINT)

The ASCII value converted to Dint value.

**Remarks**

The value at the SRC[] array is separate ASCII value.

Ex: SRC[0]- has the first ASCII value, Q – ASCII converted to Dint value.

ST Language

Q := AsciiToDint(Src[]);

FBD Language

**LD Language**

IL Language

Not Available.

**See also**

[AsciiToInt](#) [AsciiToReal](#)

## AsciiToInt

*Operator* – Perform conversion of ASCII to a Int value.

### Inputs

SRC[ ] : (TYPE : USINT[])

SRC is the array of ASCII value placed at the input.

### Outputs

Q : (TYPE : INT)

The ASCII value converted to Int value.

### Remarks

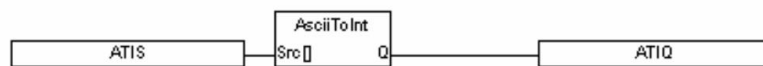
The value at the SRC[] array is separate ASCII value.

Ex: SRC[0]- has the first ASCII value, Q – ASCII converted to Int Value.

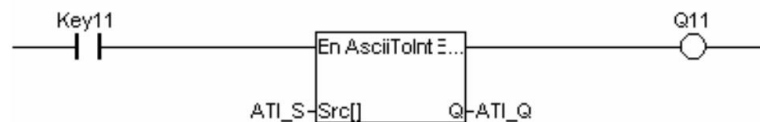
### ST Language

Q := AsciiToInt(Src[]);

### FBD Language



### LD Language



### IL Language

Not Available.

### See also

[AsciiToDint](#) [AsciiToReal](#)

## **AsciiToReal**

*Operator* – Perform conversion of ASCII to a Real value.

### **Inputs**

SRC[ ] : (TYPE : USINT[])

SRC is the array of ASCII value placed at the input.

### **Outputs**

Q : (TYPE : REAL)

The ASCII value converted to Real value.

### **Remarks**

The value at the SRC[] array is separate ASCII value.

Ex: SRC[0]- has the first ASCII value, Q – ASCII converted to Real Value.

ST Language

Q := AsciiToReal(Src[]);

**FBD Language**

**LD Language**

**IL Language**

Not Available.

### **See also**

[AsciiToDint](#)   [AsciiToInt](#)



## **DintToAscii**

*Operator* – Perform conversion of Dint to ASCII value.

### **Inputs**

SRC: (TYPE : DINT)

This is the Input in Dint format for conversion in ASCII value.

DST[ ] : (TYPE : USINT[])

The converted Dint into ASCII is placed in this array of DST.

#POINT : (TYPE : INT)

This specifies the place of the point from right of the maximum length allocated.

#MAXLEN: (TYPE : INT)

The maximum number of digits allowed for conversion.

#FILL0: (TYPE : BOOL)

Fills the vacant place after conversion from Dint to ASCII with zeroes.

### **Outputs**

Q : (TYPE : BOOL)

The output is TRUE if the Dint value is converted to ASCII successfully.

### **Remarks**

The value at the SRC are separate ASCII values.

Ex: DST[0] - has the first ASCII value & so on.

ST Language

Q := DintToAscii(Src, Dst[], #Point, #MaxLen, #Fill0);

FBD Language

### **LD Language**

IL Language

OP1: LD SRC

DINTTOASCII Dst[], #Point, #MaxLen, #Fill0

ST Q

### **See also**

[IntToAscii](#) [RealToAscii](#)

## **IntToAscii**

*Operator* – Perform conversion of Int to ASCII value.

### **Inputs**

SRC: (TYPE : INT)

This is the Input in Int format for conversion in ASCII value.

DST[ ] : (TYPE : USINT[])

The converted Dint into ASCII is placed in this array of DST.

#POINT : (TYPE : INT)

This specifies the place of the point from right of the maximum length allocated.

#MAXLEN: (TYPE : INT)

The maximum number of digits allowed for conversion.

#FILL0: (TYPE : BOOL)

Fills the vacant place after conversion from Dint to ASCII with zeroes.

### **Outputs**

Q : (TYPE : BOOL)

The output is TRUE if the Dint value is converted to ASCII successfully.

### **Remarks**

The value at the SRC are separate ASCII value.

Ex: DST[0] - has the first ASCII value & so on.

ST Language

Q := IntToAscii(Src,Dst[], #Point, #MaxLen, #Fill0);

FBD Language

**LD Language**

**IL Language**

OP1: OP1: LD SRC

INTTOASCII Dst[], #Point, #MaxLen, #Fill0

ST Q

### **See also**

[IntToAscii](#) [RealToAscii](#)

## **RealToAscii**

*Operator* – Perform conversion of Real to ASCII value.

### **Inputs**

SRC: (TYPE : REAL)

This is the Input in Real format for conversion in ASCII value.

DST[ ] : (TYPE : USINT[])

The converted Dint into ASCII is placed in this array of DST.

#POINT : (TYPE : INT)

This specifies the place of the point from right of the maximum length allocated.

#MAXLEN: (TYPE : INT)

The maximum number of digits allowed for conversion.

#FILL0: (TYPE : BOOL)

Fills the vacant place after conversion from Dint to ASCII with zeroes.

### **Outputs**

Q : (TYPE : BOOL)

The output is TRUE if the Dint value is converted to ASCII successfully.

### **Remarks**

The value at the SRC are separate ASCII value.

Ex: DST[0] - has the first ASCII value & so on.

ST Language

```
Q := RealToAscii(Src, Dst[], #Point, #MaxLen, #Fill0);
```

FBD Language

### **LD Language**

### **IL Language**

OP1: OP1: LD SRC

REALTOASCII Dst[], #Point, #MaxLen, #Fill0

ST Q

### **See also**

[DintToAscii](#) [IntToAscii](#)

## **SetString**

*Operator* – Sets the ASCII value of the source at the destination array.

### **Inputs**

SRC : (TYPE : STRING)

SRC is the source string.

DST[ ] : (TYPE :USINT[])

DST is the converted ASCII value.

### **Outputs**

Q : (TYPE :BOOL)

Output goes high if the SetString is successful.

### **Remarks**

The value at the DST[] array is character wise ASCII value.

Ex: DST[0]- has the first alphabet's ASCII value, DST[1]- has the second alphabet's ASCII value & so on.

ST Language

Q := SetString(Src, Dst[]);

FBD Language

### **LD Language**

IL Language

Op1: LD SRC

SetString Dst[]

ST Q

### **See also**

[CmpStringConst](#) [CmpStringVar](#) [StringLen](#)

## StringLen

*Operator* – Length of a given input string.

### Inputs

S[] : (TYPE : USINT[])

S is the source string ASCII value.

### Outputs

LEN : (TYPE :INT)

Length of the elements in array S.

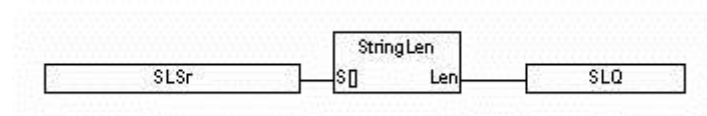
### Remarks

En is the Enable input & Eno is the enable output. En & Eno will be in same state.

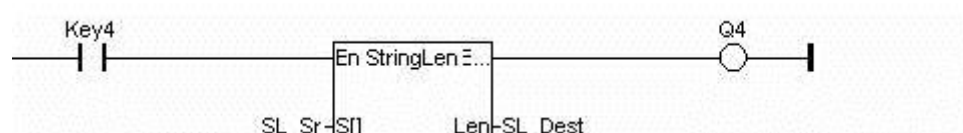
### ST Language

LEN := StringLen(S[]);

### FBD Language



### LD Language



### IL Language

Not Available.

### See also

[CmpStringConst](#) [CmpStringVar](#) [SetString](#)

## **CmpStringVar**

*Operator* – Perform comparison of String variables.

### **Inputs**

S1[ ] : (TYPE : USINT[])

S1 is the array of ASCII values placed to compare with the input at S2.

S2[ ] : (TYPE : USINT[])

S2 is the array of ASCII values to which input S1 is compared with.

#COUNT : (TYPE : INT)

The number of characters to be compared.

### **Outputs**

Q : (TYPE : BOOL)

Output goes high if the input ASCII values are equal for a given count.

### **Remarks**

Each element of array S1[] is compared with each element of array S2[].

Ex: S1[0] is compared with value at S2[0], till the given count at #COUNT input.

ST Language

Q := CmpStringVar(S1[], S2[], #Count);

FBD Language

LD Language

IL Language

Not Available.

### **See also**

[SetString](#) [CmpStringConst](#) [StringLen](#)

## **CmpStringConst**

*Operator* – Perform comparison of String with a constant.

### **Inputs**

S1[ ] : (TYPE : USINT[])

S1 is the array of ASCII values placed to compare with the input at S2.

S2: (TYPE : STRING)

S2 is the string to which input S1 is compared.

### **Outputs**

Q : (TYPE : BOOL)

Output goes high if the inputs are equal.

### **Remarks**

The value at the S1[] array is character wise ASCII value.

Ex: S1[0]- has the first alphabet's ASCII value, S1[1]- has the second alphabet's ASCII value & so on.

ST Language

Q := CmpStringConst(S1[], S2);

FBD Language

LD Language

IL Language

Not Available.

### **See also**

[SetString](#) [CmpStringVar](#) [StringLen](#)

## **CANOpen Operations**

### ***CANOpen Operations***

Below are the standard functions for managing CANOpen operations:

SDO Write	Modifies given Object Dictionary Entry.
SDO Read	Reads the given Object Dictionary Entry.
Get Local ID	Reads Local CANOpen Node ID.
Get NMT State	Read Local or Slave node CANOpen NMT state.
Set NMT State	Sets NMT state of given Node in CANOpen Network.
Receive Emergency message from a given Device	Receives Emergency message from a specific Device.
Receive Emergency message from any Device	Receives Emergency message from any Device in the network.



## SDO Read

This element reads the given object dictionary entry or the  $i^3$  register contents. If node is Master then using this block an object dictionary entry of any slave node in the network can be read. In case of Slave only own object dictionary entries can be read.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

CB[] - (Type: INT[])

This input is specified as register type and offset reference. This register is the first of three (3) registers that contain the information for the SDO read CAN message.

Word 1 Node ID – Node ID of the device (1 to 127) in case of Master only. For accessing the object dictionary /  $i^3$  register contents for self node, value 0 needs to be used.

Word 2 Index Value - Any valid 16-bit address to access the CANOpen dictionary supported by the node; for array and records the address is extended by an 8-bit sub-index.

Word 3 Sub-Index value - Any valid 8-bit sub-address to access the sub-objects of arrays and records supported by the node.

### Outputs

CBO[] - (Type: INT[])

Output is specified as a Register Type and Offset reference. This register is the first of five (5) registers that contain the information of output data.

Word 1 Kernel Error – Gives value of Kernel Error.

Word 2 SDO Error – SDO Error (Higher 16 bits)

Word 3 SDO Error – SDO Error (Lower 16 bits)

Word 4 Data Length – Amount of data to read (Max data length can be 4 bytes)

Word 5 Start of Data – Any valid Register address

### ST Language

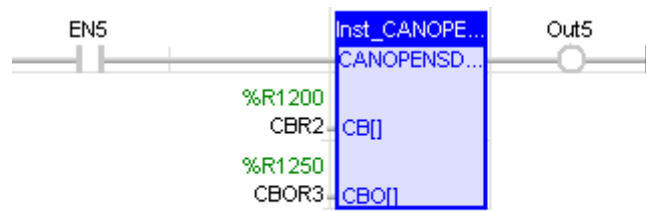
(\*Inst\_CANOPENSDOREAD is a declared instance of SDO Read Block\*)

Inst\_CANOPENSDOREAD( ENr, CBR, CBOR );

### FBD Language



### LD Language



### IL Language

BEGIN\_IL

OP2 : CAL Inst\_CANOPENSOREAD( ENR1(\*BOOL\*), CBR5(\*INT\*),  
CBOR5(\*INT\*))

END\_IL

### See also

SDO Write    Get Local ID    Get NMT State    Set NMT State

Receive Emergency message from a given Device

Receive Emergency message from any Device

## SDO Write

This element is used to change given object dictionary entry or to change the  $i^3$  register contents. If node is Master then using this block an object dictionary entry of any slave node in the network can be modified. In case of Slave only own object dictionary entries can be modified.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

CB[] - (Type: INT[])

This input is specified as register type and offset reference. This register is the first of five (5) registers that contain the information for the SDO write CAN message.

Word 1 Node ID – Node ID of the device (1 to 127) in case of Master only. For accessing the object dictionary /  $i^3$  register contents for self node, value 0 needs to be used.

Word 2 Index Value – Any valid 16-bit address to access the CANOpen dictionary supported by the node; for array and records the address is extended by an 8-bit sub-index.

Word 3 Sub-Index value – Any valid 8-bit sub-address to access the sub-objects of arrays and records supported by the node.

Word 4 Data Length - Amount of data to write (Max data length can be 4 bytes).

Word 5 Start of Data - Any valid Register address.

### Outputs

CBO[] - (Type: INT[])

Output is specified as a Register Type and Offset reference. This register is the first of three (3) registers that contain the information of output data.

Word 1 Kernel Error – Gives value of Kernel Error.

Word 2 SDO Error – SDO Error (Higher 16 bits)

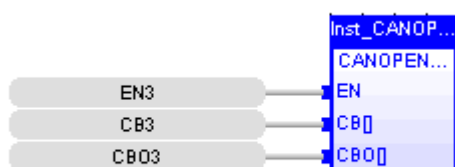
Word 3 SDO Error – SDO Error (Lower 16 bits)

### ST Language

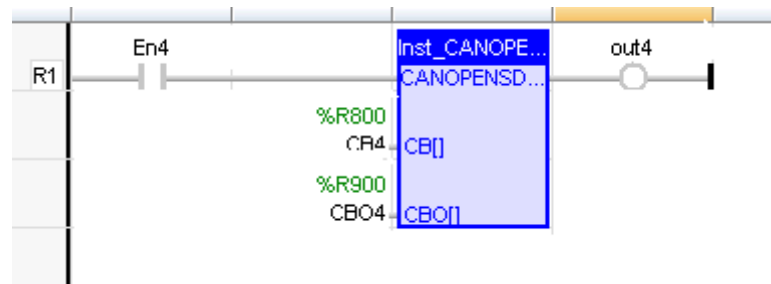
(\*Inst\_CANOPENSOWRITE is a declared instance of SDO Write Block\*)

Inst\_CANOPENSOWRITE(EN1, CB, CBO);

### FBD Language



### LD Language



### IL Language

BEGIN\_IL

OP1 : CAL CANOSDOW (EN11(\*BOOL\*), CB1(\*INT\*), CBO1(\*INT\*) )

END\_IL

### See also

SDO Read    Get Local ID    Get NMT State    Set NMT State  
Receive Emergency message from a given Device  
Receive Emergency message from any Device

### Get Local ID

This element reads the local CANOpen node ID.

Enabling the input power to this element gets the Local Node ID in the configured register.

### ST Language

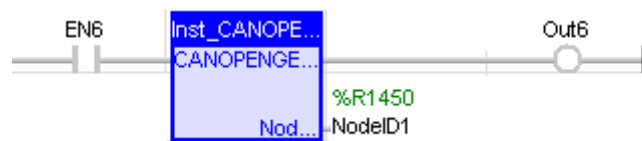
(\*Inst\_CANOPENGETLOCID is a declared instance of Get Local ID Block\*)

Inst\_CANOPENGETLOCID( ENL1 );

### FBD Language



### LD Language



### IL Language

BEGIN\_IL

OP3 : CAL Inst\_CANOPENGETLOCID( ENL2 )

END\_IL

### See also

SDO Write   SDO Read   Get NMT State   Set NMT State

Receive Emergency message from a given Device

Receive Emergency message from any Device

### Get NMT State

This element reads the local or slave node CANOpen NMT state.

#### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

NodeID - (Type: INT)

Node ID of the device for which NMT state needs to be obtained. For accessing the NMT state of self node, value 0 needs to be used.

#### Outputs

NMT State - (Type: INT)

Gives the NMT state of the device.

Note:

Following are the possible NMT states and their values in Hex:

NMT State	Value in Hex
Preoperational	7F
Operational	5
Stop	4

#### ST Language

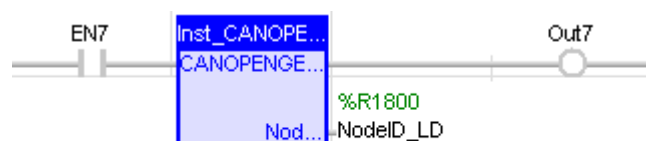
(\*Inst\_CANOPENGETSTATE is a declared instance of Get NMT State Block\*)

```
Inst_CANOPENGETSTATE( ENN1, NodeID_ST );
```

#### FBD Language



#### LD Language



## **IL Language**

BEGIN\_IL

OP4 : CAL Inst\_CANOPENGETSTATE( ENN2(\*BOOL\*), NodeID\_IL(\*INT\*) )

END\_IL

### **See also**

SDO Write   SDO Read   Get Local ID   Set NMT State

Receive Emergency message from a given Device

Receive Emergency message from any Device

## Set NMT State

This element sets NMT state of given node in the CANOpen network. This block can only be used if node is configured as Master.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

CB[] - (Type: INT[])

This input is specified as register type and offset reference. This register is the first of two (2) registers that contain the information of input data.

Word1 Node ID - Node ID of the device. For setting the NMT state of self node, value 0 needs to be used.

Word 2 NMT State – NMT command of corresponding NMT State.

Note:

Following are the NMT commands for corresponding NMT states:

States	NMT Commands
Preoperational	80
Operational	1
Stop	2
Reset Comm	81
Rest Node	82

### Outputs

KernelError - (Type: INT)

Indicates any kernel error.

### ST Language

(\*Inst\_CANOPENSETNMT is a declared instance of Set NMT State Block\*)

Inst\_CANOPENSETNMT( ENS, CBS );

### FBD Language



### LD Language



### IL Language



```
BEGIN_IL  
OP5 : CAL Inst_CANOPENSETNMT( EN_IL(*BOOL*), CB_IL(*INT*) )  
END_IL
```

**See also**

SDO Write   SDO Read   Get Local ID   Get NMT State  
Receive Emergency message from a given Device  
Receive Emergency message from any Device

## Receive Emergency Message From a Given Device

This element is used to receive emergency message from specific device.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed.

NodeID - (Type: INT)

Node ID of the device for which emergency message needs to be received

### Outputs

CBO[] - (Type: INT[])

Output is specified as a Register Type and Offset reference. This register is the first of five (5) registers that contain the information of output data.

Word1 Kernel error – Gives value of Kernel Error.

Word2 EMC Value in 64 bit – Gives the emergency message/codes.

Word3 – Lower 8 bit represents Error register and upper 8 Bit Manufacturer specific error field.

Word4 – Higher 16 bits of Manufacturer specific error.

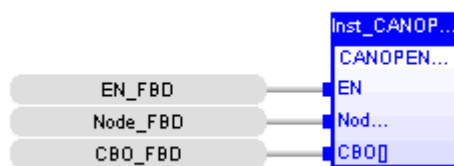
Word5 – Lower 16 bits of Manufacturer specific error.

### ST Language

(\*Inst\_CANOPENRCMCDEV is a declared instance of Receive Emergency Message from a given Device Block\*)

```
Inst_CANOPENRCMCDEV( EN_ST, Node_ST, CBO_ST );
```

### FBD Language



### LD Language



### IL Language

```
BEGIN_IL
OP6 : CAL Inst_CANOPENRCMCDEV( EN_IL1(*BOOL*), Node_IL(*INT*),
CBO_IL(*INT*) )
END_IL
```

**See also**

SDO Write   SDO Read   Get Local ID   Get NMT State   Set NMT State  
Receive Emergency message from any Device

## Receive Emergency Message From Any Device

This element is used to receive emergency message from any device in the network.

### Inputs

EN – (Type: BOOL)

The "EN" input is a condition. If EN is TRUE State then block is executed

### Outputs

**CBO[]** - (Type: INT[])

Output is specified as a Register Type and Offset reference. This register is the first of Six (6) registers that contain the information of output data.

Word1 Kernel error – Gives value of Kernel Error.

Word2 NodeID – Device Node ID.

Word3 EMC Value in 64 bit – Indicates the emergency message/codes.

Word4 – Lower 8 bit represents Error register and upper 8 Bit Manufacturer specific error field.

Word5 – Higher 16 bits of Manufacturer specific error.

Word6 – Lower 16 bits of Manufacturer specific error.

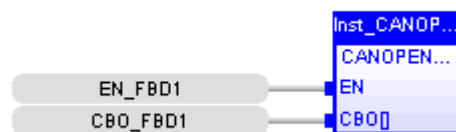
Note: Manufacturer specific error field represents IMO CANOpen Status register contents

### ST Language

(\*Inst\_CANOPENRCEMC is a declared instance of Receive Emergency Message from any Device Block\*)

```
Inst_CANOPENRCEMC( EN_ST1, CBO_ST1 );
```

### FBD Language



### LD Language



### IL Language

BEGIN\_IL

```
OP7 : CAL Inst_CANOPENRCEMC( EN_IL2, CBO_IL1 )
```

END\_IL

**See also**

SDO Write   SDO Read   Get Local ID   Get NMT State   Set NMT State  
Receive Emergency message from a given Device

## ***Error\_Details***

### **Kernel Error:**

0x0 No Kernel Error.  
0x1 Other Kernel Errors.  
0x21 SDO Engine is Busy.  
0x22 Memory Access Error.  
0x23 SDO Timeout Error.  
0x24 Not Supported.  
0x25 Emergency Busy Error.  
0x26 Input Data Error.  
0xFE Waiting for Response.  
0xFF Kernel Not Active

### **SDO Error:**

0x5040002 Invalid Block Size.  
0x5040003 Invalid Sequence Count.  
0x6010000 Index Not Supported.  
0x6010002 Index Read Only.  
0x6070012 Data Length is high.  
0x6070013 Data Length is Low.  
0x8000022 Invalid NMT State.  
0x6040041 PDO Mapping Failed.  
0x6040042 PDO Length Exceeded.  
0x6090011 Sub Index Not Supported.  
0x6090030 Value Range Exceeded.  
0x6040043 Parameter Incompatible

### **Error register Bit details**

Bit	M/O	Meaning
0	M	generic error
1	O	current
2	O	voltage
3	O	temperature
4	O	communication error (overrun, error state)
5	O	device profile specific
6	O	Reserved (always 0)
7	O	manufacturer specific

## Emergency Codes

Error Code (hex)	Meaning
00xx	Error Reset or No Error
10xx	Generic Error
20xx	Current
21xx	Current, device input side
22xx	Current inside the device
23xx	Current, device output side
30xx	Voltage
31xx	Mains Voltage
32xx	Voltage inside the device
33xx	Output Voltage
40xx	Temperature
41xx	Ambient Temperature
42xx	Device Temperature
50xx	Device Hardware
60xx	Device Software
61xx	Internal Software
62xx	User Software
63xx	Data Set
70xx	Additional Modules
80xx	Monitoring
81xx	Communication
8110	CAN Overrun (Objects lost)
8120	CAN in Error Passive Mode
8130	Life Guard Error or Heartbeat Error
8140	recovered from bus off
8150	Transmit COB-ID collision
82xx	Protocol Error
8210	PDO not processed due to length error
8220	PDO length exceeded
90xx	External Error
F0xx	Additional Functions
FFxx	Device specific

See also

SDO Write   SDO Read   Get Local ID   Get NMT State   Set NMT State  
Receive Emergency message from a given Device  
Receive Emergency message from any Device



## Screen Operations

### *Screen Operations*

Below are the standard operators that perform Screen Operations:

Change	Changes a screen
Screen	Displays a screen
Display	
Screen	

## ChangeScreen

*Operator-* Changes a screen.

### Inputs

#NUMBER - (TYPE: DINT)

#Number format means can able to enter integer value only.

### Outputs

Q - (TYPE: DINT)

Register should be in R type only.

### Remarks

Change Screen does not force screen to remain active. Operator may choose to change screen after using various navigation methods (menus, screen jumps, scrolling...).

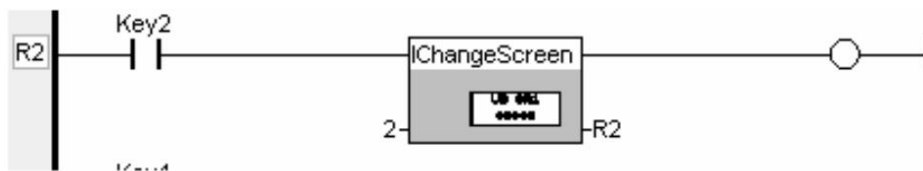
Only one change screen to be active at a time. If a more then one has no affect, however writing directly to %SR1 will change the screen.

### ST Language

OUT := ChangeScreen(#NUMBER);

### FBD Language

### LD Language



**Note:** Power does not flow through the display coil.

### IL Language

Op1: LD #NUMBER  
ChangeScreen  
ST Q

### See also

DisplayScreen

## **DisplayScreen**

Operator - Displays a screen.

### **Input**

#Number - (TYPE: DINT)

Screen number that is to be displayed. It can take integer value only.

### **Output**

Q - (TYPE: DINT)

Output Register should be in R type only.

### **Remarks**

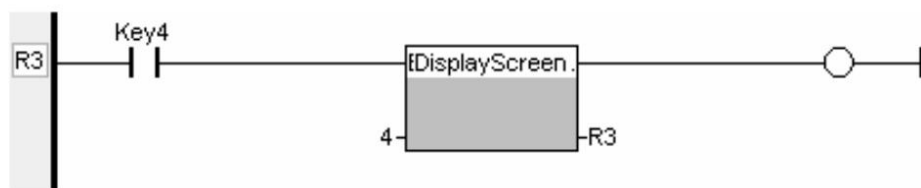
Display Screen will override any other user screen being displayed. If more than one display screen is active at a time, the last one in the ladder program is displayed. When a screen is being forced, it can be read from System Register %SR2.

### **ST Language**

Q := DisplayScreen(#NUMBER);

### **FBD Language**

### **LD Language**



**Note:** Power does not flow through the display coil.

### **IL Language**

```
Op1: LD #NUMBER
      DisplayScreen
      ST Q
```

### **See also**

ChangeScreen

## Serial Operations

### *Serial Operations*

Below are the standard operators that perform Serial Communication operations:

Close Communication Port	Close communication Port
Modbus Master	Configures Port as Modbus Master
Modbus Slave	Configures Port as Modbus Slave
Modbus Slave with Exception	Configures Port as Modbus Slave with Exception Message
Modem Auto Answer	Configures modem to auto answer a call after predefined rings
Modem Auto Dial	Configures dialing to a modem with the supplied phone No.
Modem Send Init String	Checks a Modem by sending the initialization string
Open Communication Port	Opens a desired port for communication
Open Flexible Comm. Port	Opens a flexible port for communication
Receive Data	Receives data through an opened port
Send Data	Sends data through an opened port

CloseComm

*Operator* - Performs the closure of the channel of the selected port.

### **Inputs**

EN : Enable input (TYPE : BOOL)

#PORT: Port number (TYPE : DINT)

### **Outputs**

Q : Output if the port closure is successful / Port already closed. (TYPE : BOOL)

If you attempt to close a port that does not exist, power flow through the element is FALSE.

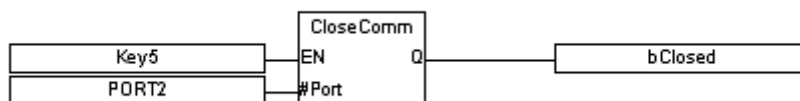
### **Remarks**

This element closes the channel to the selected port. There are no operational parameters except the Port Number. This entry must be a decimal constant.

### **ST Language**

Q := CloseComm(EN, #Port);

### **FBD Language**



### **LD Language**

### **IL Language**

```
Op1 : LD EN
      CloseComm #PORT
      ST Q
```

### **See also**

Ope  
nCo  
mm  
Rc  
vCo  
mm

ModbusMaster

*Operator* – Configures a opened port as Modbus master.

### **Inputs**

**EN** : Enable input (TYPE : BOOL)

**#Port**: (TYPE : DINT)

The communication port previously open by the ladder program with **Protocol** set to Modbus ASCII or Modbus RTU.

**Timeout**: (TYPE : INT)

Specified as either a register reference, or as a decimal constant (with a range of 0 to 1023). This specifies the amount of time that is allowed between a Modbus command and its response. This parameter is in terms of 100 milliseconds (i.e., 100 = 10.0 Sec).

**Trigger**: (TYPE : BOOL)

Specified as a bit register reference. When this bit goes from an off to on transition, the block transmits the Modbus message defined by the message control block ( MCB). When this input is low, the status word is cleared.

**MCB[ ]**: Message Control Box (TYPE : INT[])

Specified as a register reference. This register is the first of six (6) registers that contain the control information for this block.

### **Outputs**

Status: (TYPE : INT)

A WORD (16-bit) register used to hold the results of the element.

### **Remarks**

#### **1) MCB: Message Control Box**

**Word 1 Slave ID** - value from 1 to 247 indicating the device to receive the message

**Word 2 Modbus Command** - Modbus command to send to the slave

**Word 3 Slave Offset** - Starting point in the Modbus slave for data to read or write  
- 1

**Word 4 Data Length** - Amount of data to read or write

**Word 5 Controller Reference Type** - Enumerated controller register type

**Word 6 Controller Reference Offset** - Controller register number - 1

#### **2) Status bit assignment:**

##### **Bit Number Status**

- |   |  |
|---|--|
| 1 | Request Succeeded (OK)                       |
| 2 | Request Failed (See additional errors below) |
| 3 | ID out of range                              |
| 4 | Length exceeds Modbus frame                  |

5	Command not supported
6	Invalid controller reference
7	Reserved
8	Reserved
9	Timeout Expired
10	Frame or parity error
11	Invalid checksum / crc from slave
12	Invalid format from slave
13	Slave rejected the command
14	Slave rejected the address
15	Slave rejected the data
16	Slave device error

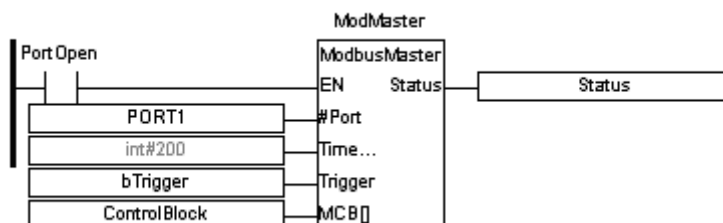
This function passes power flow if the associated port is opened and ready for communications.

### **ST Language**

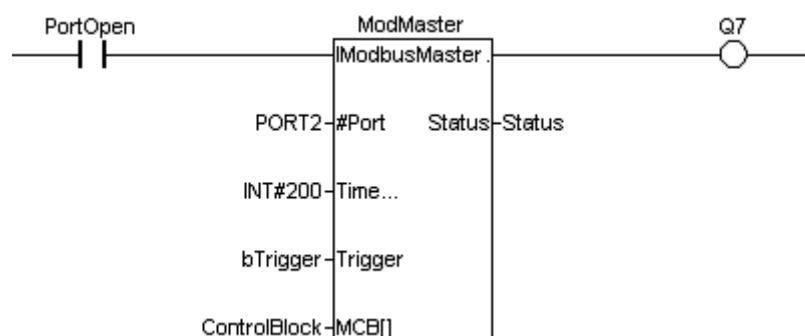
(\* ModMaster is a declared instance of ModbusMaster function block \*)

```
ModMaster(EN, #PORT, Timeout, Trigger, MCB[]);
status := ModMaster.Status;
```

### **FBD Language**



### **LD Language**



### **IL Language**

(\* ModMaster is a declared instance of ModbusMaster function block \*)

```
OP1: CAL ModMaster(EN, #PORT, Timeout, Trigger, MCB[])
      LD ModMaster.Status
```

## ST STATUS

### ***See also***

M  
o  
d  
b  
u  
s  
S  
l  
a  
v  
e



## ModbusSlave

*Operator* – Configures a opened port as Modbus Slave.

### Inputs

**EN:** Enable input (TYPE : BOOL)

**#Port:** (TYPE : DINT)

The communication port previously open by the ladder program with **Protocol** set to Modbus ASCII or Modbus RTU.

**Address:** (TYPE : INT)

This specifies either a register or as a decimal constant (with a range of 1 to 247). This specifies the Modbus address the controller uses to respond to Modbus request

**Time:** (TYPE : INT)

This specifies either a register or as a decimal constant (with a range of 0 to 1023).

This specifies the amount of time that may pass between request from the master before the in-activity timeout bit is set in the status word. This parameter is in terms of 100 milliseconds (i.e., 100 = 10.0 Sec).

### Outputs

**Status:** (TYPE : INT)

A WORD (16-bit) register used to hold the results of the element.

### Remarks

Status bit assignment:

**Bit Number    Status**

1	Inactivity Timeout
4	Valid message received (toggles)
5	Parity error (single pass)
6	Frame Error (single pass)
7	Overrun error (single pass)
8	Crc/Checksum error (single pass)

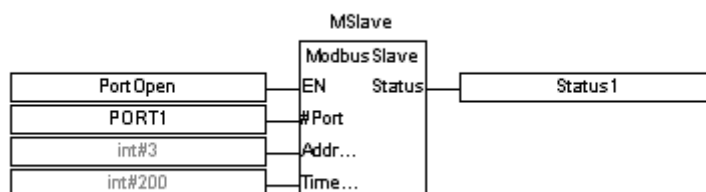
### ST Language

(\* ModSlav is a declared instance of ModSlave function block \*)

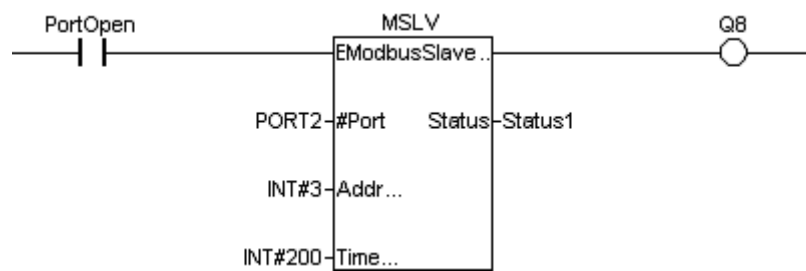
ModSlav(EN, #PORT, Address, Time);

Status := ModSlav.Status;

### FBD Language



### LD Language



### ***IL Language***

(\* ModSlav is a declared instance of ModSlave function block \*)

```
OP1 : CAL ModSlav(EN, #PORT, Address, Time)
      LD ModSlav.Status
      ST status
```

### ***See also***

M  
o  
d  
b  
u  
s  
M  
a  
s  
t  
e  
r

## ModbusSlaveEx

*Operator* – Configures a opened port as Modbus Slave with Exception Message.

### Inputs

**EN:** Enable input (TYPE : BOOL)

**#Port:** (TYPE : DINT)

The communication port previously open by the ladder program with **Protocol** set to Modbus ASCII or Modbus RTU.

**Address:** (TYPE : INT)

This specifies either a register or as a decimal constant (with a range of 1 to 247). This specifies the Modbus address the controller uses to respond to Modbus request.

**Time:** (TYPE : INT)

This specifies either a register or as a decimal constant (with a range of 0 to 1023).

This specifies the amount of time that may pass between request from the master before the in-activity timeout bit is set in the status word. This parameter is in terms of 100 milliseconds (i.e., 100 = 10.0 Sec).

**COUNT:** (TYPE : INT)

This specifies as a register. This contains the number of bytes in the Message Data buffer to send. Transition from zero to a non-zero value triggers the transmission of one Exception Message.

**DATA:** (TYPE : INT)

This specifies as a register. This is the first register number of an array, which contains the Exception Message.

### Outputs

Status: (TYPE : INT)

A WORD (16-bit) register used to hold the results of the element.

### Remarks

Status bit assignment:

Bit Number	Status
1	Inactivity Timeout
4	Valid message received (toggles)
5	Parity error (single pass)
6	Frame Error (single pass)
7	Overrun error (single pass)
8	Crc/Checksum error (single pass)
9	Exception message send (reset when e_cnt = 0)
10	Exception message exceeds send buffer size (reset when e_cnt = 0)
11	Attempt to send exception message when transmit busy (reset when e_cnt = 0)

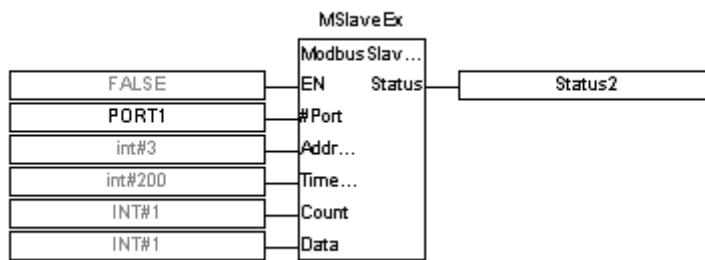
### ST Language

(\* ModSlavEx is a declared instance of ModSlaveEx function block \*)

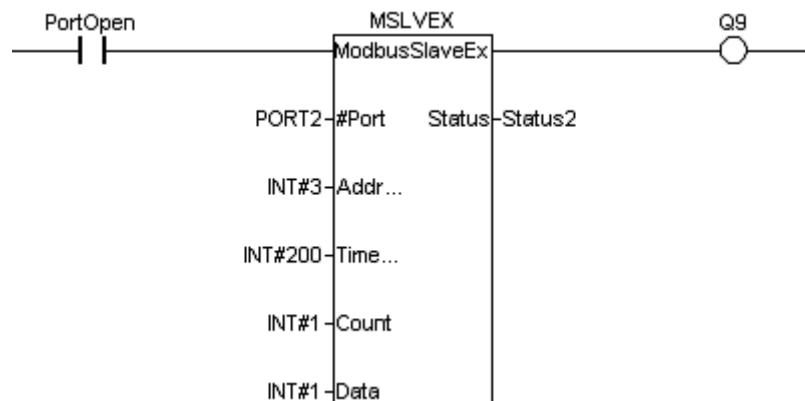
ModSlavEx(EN, #PORT, Address, Time, Count, Data);

Status := ModSlavEx.Status;

## FBD Language



## LD Language



## IL Language

(\* ModSlavEx is a declared instance of ModSlaveEx function block \*)

```
OP1 : CAL ModSlavEx(EN, #PORT, Address, Time, Count, Data)
      LD ModSlavEx.Status
      ST status
```

## See also

[ModbusMaster](#) [ModbusSlave](#)

## **ModemAutoAnswer**

*Operator* – Places the specified modem in Auto-answer mode. The modem will answer incoming calls after the specified number of rings. Set the number of rings to zero to disable auto answer mode on the modem.

### **Inputs**

**EN** : Enable input (TYPE : BOOL)

**#Port**: (TYPE : DINT)

The communication port opened.

**#NbRing**: (TYPE : DINT)

Number of Rings, after which the modem answers the call.

### **Outputs**

Status: (TYPE : INT)

Status indicates the progress and success or failure of the operations.

### **Remarks**

#### **1) General Status Results**

The modem function block returns a status word to indicate the progress and success or failure of the operations. All Actions share this result information:

- 1 When the function is not active, output from the function block is OFF.
- 2 When the function is being executed, the output remains OFF.
- 3 When a function times-out, because the modem did not respond, the output remains OFF.
- 0 The modem accepted the command, the output depends on the function.
  - 1. The modem successfully connected, the output turns ON.
  - 2. When an incoming ring is detected, the output remains OFF.
  - 3. The modem loses carrier, the output turns OFF.
  - 4. When the command results in an error, the output remains OFF

#### **2) Action**

When the input to this function block transitions from OFF to ON, the controller attempts to set the modem to auto answer mode. When the modem has been placed in auto answer mode, the function block output remains OFF, and the status is loaded with **0** to indicate the modem is OK. Power flow into the function block must be kept ON at this point. When an incoming call is received, the output of the function block remains OFF, and the status is loaded with **2** to indicate ringing. After the programmed number of rings, the modem answers. If a connection is established, the output of the function block turns ON and the status is loaded with **1** to indicate a connection. If the connection is lost, (other side hangs-up ...) the output turns OFF and the status is loaded with **3** to indicate a loss of carrier.

If the modem is connected to another device and the input to the function block transitions from ON to OFF, the modem hangs-up, the function block output turns OFF, and the status is loaded with **-1** to show the function block is inactive.

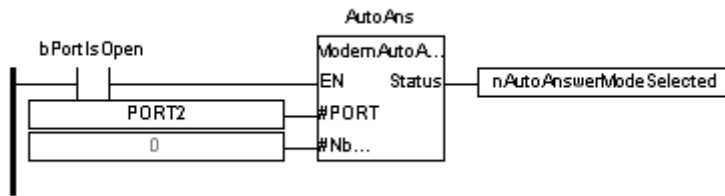
### **ST Language**

(\* AutoAns is a declared instance of ModemAutoAnswer function block \*)

AutoAns(EN, #PORT, #NbRing);

Status := AutoAns.Status;

## ***FBD Language***



## ***LD Language***

### ***IL Language***

(\* AutoAns is a declared instance of ModemAutoAnswer function block \*)

```
Op1: CAL AutoAns(EN, #PORT, #NbRing)
      LD AutoAns.Status
      ST STATUS
```

### ***See also***

[ModemAutoDial](#)   [ModemSendInit](#)

## **ModemAutoDial**

*Operator* – Perform dialing to a modem with the supplied phone number.

### **Inputs**

**EN** : Enable input (TYPE : BOOL)

**#Port**: (TYPE : DINT)

The communication port opened.

**#Option**: (TYPE : INT)

Option is the command your modem requires to initiate dialing.

**Str**: (TYPE : STRING)

Str is the number to be dialed. Do not use spaces, dashes, or any other punctuations except those required by the modem.

### **Outputs**

Status: (TYPE : INT)

Status indicates the progress and success or failure of the operations.

### **Remarks**

#### **1) Action**

When the input to this function block transitions from OFF to ON, the controller attempts to dial the modem with the supplied phone number. If the modem dials and successfully connects to another modem, the output of the function block turns ON, and status is loaded with **1** to indicate a connection. Keeping power flow to the dial function block allows the controller to monitor the connection. If the connection is lost (the other side hangs-up...), the output to the function block turns OFF and the status is loaded with **3** to indicate a loss of carrier.

If the input to this function block transitions from ON to OFF, the modem hangs up, the function block output turns OFF, and the status is loaded with **-1** to show the function block is inactive.

#### **2) Number to be dialed**

Specifies the phone number to be dialled. Dependant on the modem used it is also possible to include special commands in the dial string, such as those necessary to insert a pause or to defeat Call Waiting. Typically a comma "," may be used to insert a pause, for example when it is required to obtain an outside line.

Examples:

· **9,7654321** Dials for an outside line, pauses to make the connection, then dials local number 765-4321

· **\*7013175551212** Disables Call Waiting, then dials long distance to (317)555-1212.

#### **3) Option**

The above command is modem specific. The values default to common commands used to program modems using the industry standard AT command set. Most current modems implement this command structure, but refer to the User Manual that comes

with the modem in order to determine the exact strings necessary to perform these tasks.

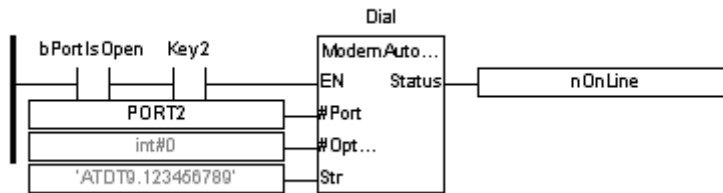
### **ST Language**

(\* AutoDial is a declared instance of ModemAutoDial function block \*)

```
AutoDial(EN, #PORT, #Option, Str);
```

```
Status := AutoDial.Status;
```

### **FBD Language**



### **LD Language**

#### **IL Language**

(\* AutoDial is a declared instance of ModemAutoDial function block \*)

```
Op1: CAL AutoDial(EN, #PORT, #Option, Str)
```

```
LD AutoDial.Status
```

```
ST STATUS
```

### **See also**

[ModemAutoAnswer](#) [ModemSendInit](#)



## **ModemSendInit**

*Operator* – Perform checking of Modem by sending the initialization string

### **Inputs**

**EN** : Enable input (TYPE : BOOL)

**#PORT**: (TYPE : DINT)

The communication port opened.

**INIT**: (TYPE : STRING)

Initialization String.

### **Outputs**

Status: (TYPE : INT)

Status indicates the progress and success or failure of the operations.

### **Remarks**

#### **1) Action**

When the input to this function block transitions from OFF to ON, the controller sends the initialization string to the modem. If the modem returns an **OK** response, the output of the function block turns ON, and the status register is loaded with **0**. If the modem returns an error response because the init string is not valid, the status register is loaded with a **4**. If the modem does not respond, the status is loaded with **-3** indicating a timeout.

### **ST Language**

(\* ModSIN is a declared instance of ModSendInit function block \*)

ModSIN(EN, #PORT, INIT);

Status := ModSIN.Status;

### **FBD Language**

### **LD Language**

### **IL Language**

(\* ModSIN is a declared instance of ModSendInit function block \*)

Op1 : CAL ModSIN(EN, #PORT, INIT)

LD ModSIN.Status

ST STATUS

### **See also**

**ModemAutoAnswer**   **ModemAutoDial**

## OpenComm

*Operator* - Performs the opening of a desired port for communication.

### Inputs

EN : Enable input (TYPE : BOOL)  
#Port: Port number (TYPE : DINT)  
#Baud: Baud setting (TYPE : DINT)  
#Parity: Parity setting (TYPE : DINT)  
#Data: Data bits (TYPE : DINT)  
#Stop: Stop bits (TYPE : DINT)  
#Handshake: Handshake (TYPE : DINT)  
#Protocol: Protocol setting (TYPE : DINT)  
#Mode: Mode of communication (TYPE : DINT)

### Outputs

Q : Output if the port open is successful (TYPE : BOOL)

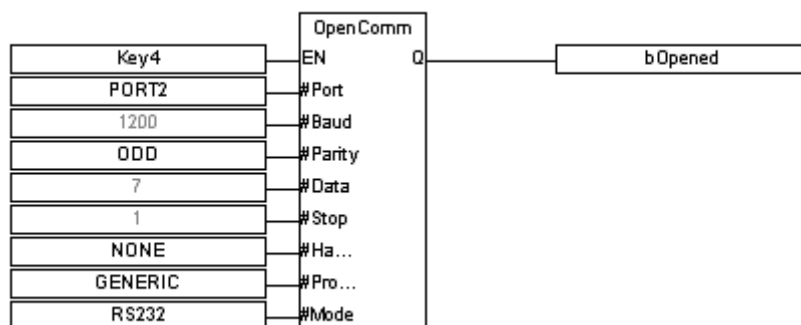
### Remarks

The Open Port element creates a channel to the desired communication port. The operational parameters (baud rate, etc) are also set by this element. The channel remains open until closed by the Close Port element or the controller is taken out of RUN mode.

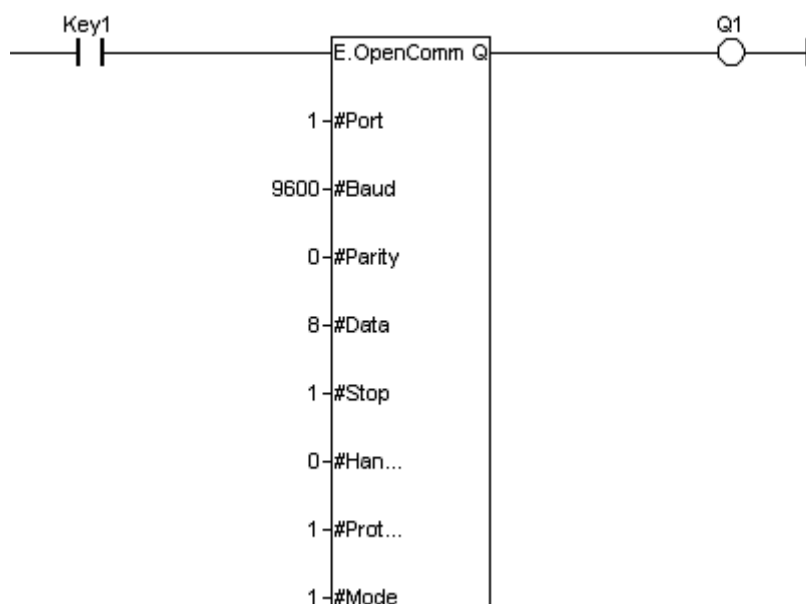
### ST Language

```
Q := OpenComm(EN, #Port, #Baud, #Parity, #Data, #Stop, #Handshake, #Protocol, #Mode);
```

### FBD Language



### LD Language



### **IL Language**

OP1 : LD EN

OpenComm #Port, #Baud, #Parity, #Data, #Stop, #Handshake, #Protocol,  
#Mode  
ST Q

### **Operands:**

**The following values can be fed to different Parameters of the block:**

1. #Ports : Ports selection (Values 0 - 2)

Port 1 (MJ1) = 0

Port 2 (MJ2) = 1

Port 3 (CN1) = 2

2. #Baud : The Baud should be the same as that of the other device it is communicating to.

BAUD\_300, // 0 = 300 baud

BAUD\_600, // 1 = 600 baud

BAUD\_1200, // 2 = 1200 baud

BAUD\_2400, // 3 = 2400 baud

BAUD\_4800, // 4 = 4800 baud

BAUD\_9600, // 5 = 9600 baud

BAUD\_19200, // 6 = 19200 baud

BAUD\_38400, // 7 = 38400 baud

BAUD\_57600, // 8 = 57600 baud

BAUD\_115200 // 9 = 115200 baud

BAUD\_14400 // 10 = 14400 baud

BAUD\_28800 // 11 = 14400 baud

BAUD\_10400 // 12 = 14400 baud

**3. #Parity : Parity can be configured as 0- 4**

PARITY\_NONE 0 = No parity

PARITY\_ODD 1 = Odd parity

PARITY\_EVEN 2 = Even parity  
PARITY\_MARK 3 = Mark parity  
PARITY\_SPACE 4 = Space parity

**4. #Data : Data bit can be 5 to 8**

DATA bits 0 = 5 data bits  
DATA bits 1 = 6 data bits  
DATA bits 2 = 7 data bits  
DATA bits 3 = 8 data bits

**5. #Stop : Stop bits can be configured as 1 or 2**

STOP\_1, // 0 = 1 stop bit  
STOP\_2, // 1 = 2 stop bits

**6. #Handshake : Handshake can be configured as 0 to 5**

HANDSHAKE\_NO, // 0 = No handshaking  
HANDSHAKE\_SW, // 1 = Software handshaking ( XON/ XOFF)  
HANDSHAKE\_HW, // 2 = Hardware handshaking ( RTS/ CTS)  
HANDSHAKE\_MD\_FD, // 3 = Multidrop full-duplex handshaking  
HANDSHAKE\_MD\_HD, // 4 = Multidrop half-duplex handshaking  
HANDSHAKE\_RM // 5 = Radio modem handshaking

**7. #Protocol : Protocol can be configured as 0 to 5**

PROTOCOL\_RISM, // 0 = RISM protocol  
PROTOCOL\_CsCAN, // 1 = iCAN protocol (not supported)  
PROTOCOL\_GENERIC, // 2 = Generic ladder-driven protocol  
PROTOCOL\_RTU, // 3 = Modbus RTU ladder-driven protocol  
PROTOCOL\_ASCII // 4 = Modbus ASCII ladder-driven protocol  
PROTOCOL\_MODBUS\_TCP // 5 = Modbus TCP ladder - driven protocol

**8. #Mode : Mode can be configured as 0 to 2**

PORT\_RS232, // 0 = Enable RS232 port  
PORT\_RS485 // 1 = Enable RS485 port PORT\_OPTION1  
MODEM // 2 = MODEM  
ETHERNET // 3 = Ethernet  
Fiber A // 4 = Fiber A  
Fiber B // 5 = Fiber B  
GSM Dual // 6 = GSM Dual  
GSM Quad // 7 = GSM Quad  
Radio 900MHZ // 8 = Radio 900Mhz  
Radio Zigbee // 9 = Radio Zigbee

**Note:**

Modbus RTU forces 8 data bits  
Modbus ASCII forces 7 data bits

**See also**

C  
l  
o  
s  
e  
C  
o  
m  
m

R  
c  
v  
C  
o  
m  
m

## **RcvComm**

*Operator* - Performs receiving of data through an opened port.

### **Inputs**

**EN** : Enable input (TYPE : BOOL)

**#PORT**: Port number (TYPE : DINT)

Port number is the comm port previously open by the ladder program.

**N** : (TYPE : INT)

This value indicates the number of bytes to be received.

**DATA[ ]** : (TYPE : USINT[])

Data is the address where the received data is to be stored. This must be specified as a register.

### **Outputs**

**COUNT** : (TYPE : INT)

Contains the number of bytes that have been copied from the port's internal buffer to the registers at **DATA** (or -1 when the function is not active).

### **Remarks**

If the port is not opened the Receive Element does nothing, and power flow through the element is FALSE.

Power flow through the element is FALSE until the requested number of characters has been received from the comm port buffer (at which time the power flow will be TRUE). It is possible that the element can not transfer all data in one program scan time, especially at slower baud rates.

The Input N can be a register references. The maximum acceptable value is 255 bytes. When the register contains a value less than 0 (zero) or greater than 255, the element does nothing, and power flow through the element is FALSE.

### **ST Language**

(\* RCVCOM is a declared instance of RCVCOMM function block \*)

RCVCOM(EN, #Port, N, DATA[]);

COUNT := RCVCOM.COUNT;

### **FBD Language**

### **LD Language**

### **IL Language**

(\* RCVCOM is a declared instance of RCVCOMM function block \*)

OP1 : CAL RCVCOM(EN, #Port, N, DATA[])

LD RCVCOM.Count

ST COUNT

### **See also**

Op  
en  
Co

m  
m  
C  
los  
eC  
om  
m

## **SendComm**

*Operator* - Performs transmitting of data through a opened port.

### **Inputs**

**EN:** Enable input (TYPE : BOOL)

**#PORT:** Port number (TYPE : DINT)

This is comm port previously open by the ladder program.

**N:** (TYPE : INT)

This value indicates the number of bytes to be transmitted, may be specified as either a register reference or as a decimal constant.

**DATA[ ]:** (TYPE : USINT[])

This is the address of the buffer containing the data to be sent. This must be specified as a Register Type and Offset reference.

### **Outputs**

**COUNT:** (TYPE : INT)

The actual number of bytes transferred to the port's internal buffer (or -1 when the function is not active). This location must be specified as a register Type and Offset reference.

### **Remarks**

If the port has been successfully opened, this element sends a specified number of bytes to the internal transmit buffer for the selected communication port.

When power is applied to the element, the Output COUNT register contains the number of characters actually transferred to the comm port transmit buffer. If power is not applied to the element, this register contains **-1** (negative one).

Power flow through the element is FALSE until the requested number of characters have been transferred to the comm port transmit buffer (at which time the power flow is TRUE). It is possible that the element can not transfer all data in one program scan time.

If the port is not open, the Transmit Element does nothing, and power flow through the element is FALSE.

If the value contained in BYTES is greater than 255, the element does nothing and power flow through the element is FALSE.

The Number of Bytes can be either a register references or a decimal constant. The maximum acceptable value is 255 bytes. When using a Register Type if it contains a value less than 0 (zero) or greater than 255, the element does nothing, and power flow through the element is FALSE.

### **ST Language**

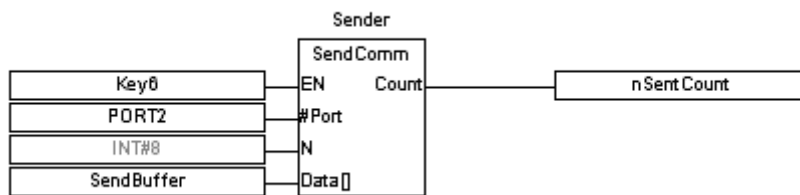
(\* SNDC is a declared instance of SendComm function block \*)

```
SNDC(EN, #PORT, N, Data[]);
```

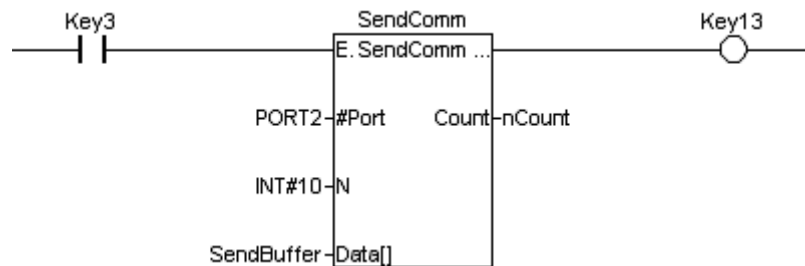
```
Count := SNDC.count;
```



## FBD Language



## LD Language



## IL Language

(\* SNDC is a declared instance of SendComm function block \*)

```
Op1 : CAL SNDC(EN, #PORT, N, Data[])  
      LD SNDC.Status  
      ST STATUS
```

## See also

O  
p  
e  
n  
C  
o  
m  
m

## Removable Media Operations

### *Removable Media Operations*

Below are the standard operators that perform Removable Media Operations:

Delete	Deletes a file on the Removable Media card
Removable Media File	Reads a comma separated value file from the Removable Media interface into controller register space
Read Removable Media File	Reads a comma separated value file from the Removable Media interface into controller register space
Read Removable Media2	Renames a file on the Removable Media card
Rename	Writes a comma separated value file to the Removable Media interface from controller register space
Removable Media File	Writes a comma separated value file to the Removable Media interface from controller register space
Write Removable Media Write Removable Media2 Removable Media Filenames	

## **Delete\_RM**

*Operator* – Perform deletion of a file on the Removable Media card.

### **Inputs**

EN : The rung state in a LD diagram is always Boolean. Blocks are connected to the rung with their input and output. The "EN" input is a trigger condition The block is executed only if EN is TRUE (TYPE : BOOL)

FILENAME: (TYPE : STRING)

This is the filename to delete.

### **Outputs**

STATUS : (TYPE : DINT)

This is a 16-bit controller registers used to show the status of the function block. See the possible status codes in the status section below.

### **Remarks**

#### **1) FILENAME**

The filename may be either fixed or obtained by reading from a contiguous block of registers in the *i<sup>3</sup>*, and may includes sub directories (i.e. "my\_data\test.csv"), Where a fixed string is used, it may be up to 147 characters long. If the filename is obtained from *i<sup>3</sup>* registers it is still limited to 147 characters and must be terminated with a NULL (byte containing zero).

If the filename is to come from *i<sup>3</sup>* and registers and is entered as a register tag ensure thats a percent (%) symbol appears before the register name. This is used to differentiate between "R1234" which is a valid file name and "%R1234" which is a register reference.

File and directory names are limited to the old DOS 8.3 convention. This is 8 characters for the name and 3 characters for an extension with a period (.) separating them. See the Removable Media Filenames section for more filename options.

#### **2 ) Status Values Returned by Removable Media Function Blocks**

##### **ValueDescription**

0	Operation completed successfully
-1	End of file was reached before completing
-2	Function is active, waiting for operation to complete
-3	Function is waiting on another RM function to complete
-4	Function block is inactive (usually no power flow)
1	Card present but unknown format
2	No card in slot
3	Card present, but not supported
4	Card swapped before operation was complete
5	Unknown error
66	File / Path specified does not exist
73	Bad file descriptor (corrupt file)

- 77 Attempt to open / rename file that is open
- 81 Specified file already exist
- 86 Function block contains illegal parameter
- 88 Too many open files\*
- 92 Attempt to write failed
- 94 Sharing violation\*
- 95 No disk present\*
- 96 Directory structure corrupt
- 98 Incorrect data format

### **3) System Registers used with Removable Media**

**%SR175 Status** - This shows the current status of the Removable Media interface.  
Possible status values:

- 0 Removable Media Interface OK
- 1 Card Present but unknown format
- 2 No card in slot
- 3 Card present, but not supported
- 4 Card swapped before operation was complete
- 5 Unknown error

**%SR176 Free Space** - This 32-bit register shows the free space on the Removable Media card in bytes.

**%SR178 Card Capacity** - This 32-bit register shows the total card capacity in bytes.

### **ST Language**

(\* DELETE is a declared instance of DELETE\_RM function block \*)  
DELETE(EN, FILENAME);  
STATUS := DELETE.STATUS;

### **FBD Language**

#### **LD Language**

**ENO/Rung Power** - This function passes power once the Status returns a 0, assuming power is still applied to the function. Should power be lost to the Rename function before it is finished, however, the function will still complete. The block is executed only if EN is TRUE then ENO also is TRUE. (i.e.) The "ENO" output always represents the same status as the "EN" input. Data type is BOOL.

### **IL Language**

(\* DELETE is a declared instance of DELETE\_RM function block \*)  
Op1: CAL DELETE(EN, FILENAME)  
LD DELETE.STATUS  
ST STATUS

**See also**

READ\_RM READ\_RM2 WRITE\_RM WRITE\_RM2 RENAME\_RM  
DELETE\_RM

## **Rename\_RM**

*Operator* – Perform renaming a file on the Removable Media card. The data in the file is not changed.

### **Inputs**

EN : (TYPE : BOOL)

The rung state in a LD diagram is always Boolean. Blocks are connected to the rung with their input and output. The "EN" input is a trigger condition The block is executed only if EN is TRUE.

OLDNAME: (TYPE : STRING)

This is the original filename to rename.

NEWNAME: (TYPE : STRING)

This is the new filename. This can be a constant or a controller registers and has the same requirements as the old filename.

### **Outputs**

STATUS : (TYPE : DINT)

This is a 16-bit controller registers used to show the status of the function block. The first 16-bit register is a status code. See the possible status value in the status section below.

### **Remarks**

#### **1) OLDNAME**

The filename may be either fixed or obtained by reading from a contiguous block of registers in the *i<sup>3</sup>*, and may includes sub directories (i.e. "my\_data\test.csv"), Where a fixed string is used, it may be up to 147 characters long. If the filename is obtained from *i<sup>3</sup>* registers it is still limited to 147 characters and must be terminated with a NULL (byte containing zero).

If the filename is to come from *i<sup>3</sup>* and registers and is entered as a register tag ensure thats a percent (%) symbol appears before the register name. This is used to differentiate between "R1234" which is a valid file name and "%R1234" which is a register reference.

File and directory names are limited to the old DOS 8.3 convention. This is 8 characters for the name and 3 characters for an extension with a period (.) separating them. See the Removable Media Filenames section for more filename options.

#### **2) Status Values Returned by Removable Media Function Blocks**

##### **ValueDescription**

- |    |  |
|----|--|
| 0  | Operation completed successfully                       |
| -1 | End of file was reached before completing              |
| -2 | Function is active, waiting for operation to complete  |
| -3 | Function is waiting on another RM function to complete |

- 4     Function block is inactive (usually no power flow)
  
- 1     Card present but unknown format
- 2     No card in slot
- 3     Card present, but not supported
- 4     Card swapped before operation was complete
- 5     Unknown error
  
- 66    File / Path specified does not exist
- 73    Bad file descriptor (corrupt file)
- 77    Attempt to open / rename file that is open
- 81    Specified file already exist
- 86    Function block contains illegal parameter
- 88    Too many open files\*
- 92    Attempt to write failed
- 94    Sharing violation\*
- 95    No disk present\*
- 96    Directory structure corrupt
- 98    Incorrect data format

### ***ST Language***

(\* RENAME is a declared instance of RENAME\_RM function block \*)

```
RENAME(EN, OLDNAME, NEWNAME);
STATUS := RENAME.STATUS;
```

### ***FBD Language***

### ***LD Language***

**ENO/Rung Power** - This function passes power once the Status returns a 0, assuming power is still applied to the function. Should power be lost to the Rename function before it is finished, however, the function will still complete. The block is executed only if EN is TRUE then ENO also is TRUE. (i.e.) The "ENO" output always represents the same status as the "EN" input. Data type is BOOL.

### ***IL Language***

```
(* RENAME is a declared instance of RENAME_RM function block *)
OP1: CAL RENAME(EN, OLDNAME, NEWNAME)
      LD RENAME.Status
      ST Status
```

***See also***

READ\_RM READ\_RM2 WRITE\_RM WRITE\_RM2 RENAME\_RM  
DELETE\_RM



## **Read\_RM**

*Operator* – Perform reading from a fixed File name on the Removable Media interface.

### **Inputs**

EN : Enable input (TYPE : BOOL)

#TYPE: (TYPE : DINT)

This is the type of data that is read.

FILENAME: (TYPE : STRING)

This is the fixed filename to read the values from and enter into the controller.

OFFSET : (TYPE : DINT)

This parameter defines where in the file to start reading data. This can be a constant value or a 32-bit controller registers.

NUM: (TYPE : INT)

This determines the number of element to read. It should be a constant. Data Type is INT.

DEST[ ]: (TYPE : ANY[])

This is a controller register where the read data is placed. Because each element can require more than one 16-bit registers

(DINT, REAL, UDINT, ASCII types) and more than one element can be read at a time this can fill a large number of registers from this starting point.

#FILETYPE : (TYPE : DINT)

This defines the extension of the file name.

### **Outputs**

STATUS : (TYPE : DINT)

This is a 32-bit controller register used to show the status of the function block. The first 16-bit register is a status code; see the possible status codes in the status section below. The second 16-bit register shows the number of elements successfully read

### **Remarks**

#### **1) FILENAME**

This is a constant. It can be up to 147 characters long that includes sub directories (i.e. "my\_data\test.csv").

File and directory names are limited to the old DOS 8.3 convention. This is 8 characters for the name and 3 characters for an extension with a period (.) separating them. See the Removable Media Filenames section for more filename options.

String expressions must be written between single quote marks.

#### **2) Status Values Returned by Removable Media Function Blocks**

##### **ValueDescription**

- |    |  |
|----|--|
| 0  | Operation completed successfully                       |
| -1 | End of file was reached before completing              |
| -2 | Function is active, waiting for operation to complete  |
| -3 | Function is waiting on another RM function to complete |
| -4 | Function block is inactive (usually no power flow)     |

- 1 Card present but unknown format
- 2 No card in slot
- 3 Card present, but not supported
- 4 Card swapped before operation was complete
- 5 Unknown error

- 66 File / Path specified does not exist
- 73 Bad file descriptor (corrupt file)
- 77 Attempt to open / rename file that is open
- 81 Specified file already exist
- 86 Function block contains illegal parameter
- 88 Too many open files\*
- 92 Attempt to write failed
- 94 Sharing violation\*
- 95 No disk present\*
- 96 Directory structure corrupt
- 98 Incorrect data format

### ***ST Language***

(\* RD is a declared instance of READ\_RM function block \*)

RD (EN, #TYPE, FILENAME, OFFSET, NUM, DEST[], #FILETYPE);  
STATUS := RD.Status;

### ***FBD Language***

### ***LD Language***

### ***IL Language***

(\* RD is a declared instance of READ\_RM function block \*)

OP1 : CAL RD(EN, #TYPE, FILENAME, OFFSET, NUM, DEST[], #FILETYPE)  
LD RD.Status  
ST Status

### ***See also***

READ\_RM READ\_RM2 WRITE\_RM WRITE\_RM2 RENAME\_RM  
DELETE\_RM

## **Read\_RM2**

*Operator* – Perform reading a comma separated value file from the Removable Media interface where the filename is obtained from *i<sup>3</sup>* registers.

### **Inputs**

EN : Enable input (TYPE : BOOL)

#TYPE: (TYPE : DINT)

This is the type of data that is read.

FILENAME: (TYPE : STRING)

This is the filename to read obtained from *i<sup>3</sup>* registers.

OFFSET : (TYPE : DINT)

This parameter defines where in the file to start reading data. This can be a constant value or a 32-bit controller registers

NUM: (TYPE : INT)

This determines the number of element to read. It should be a constant. Data Type is INT.

DEST[ ]:(TYPE : ANY[])

This is a controller register where the read data is placed. Because each element can require more than one 16-bit registers (DINT, REAL, UDINT, ASCII types) and more than one element can be read at a time this can fill a large number of registers from this starting point.

#FILETYPE : (TYPE : DINT)

This defines the extension of the file name.

### **Outputs**

STATUS : (TYPE : DINT)

This is a 32-bit controller register used to show the status of the function block. The first 16-bit register is a status code; see the possible status codes in the status section below. The second 16-bit register shows the number of elements successfully read.

### **Remarks**

#### **1) FILENAME**

The filename is stored in contiguous block of *i<sup>3</sup>* registers and has a limit of 147 characters and must be terminated with a NULL (byte containing zero). A percent (%) symbol before the register name is required indicate the file is in a register. This shows that "%R1234" is a valid register reference.

File and directory names are limited to the old DOS 8.3 convention. This is 8 characters for the name and 3 characters for an extension with a period (.) separating them. See the Removable Media Filenames section for more filename options.

Data Type is USINT / BYTE: Unsigned 8 bit Integer.

Unsigned small integer constant expressions are used valid integer values (between 0 and 255) and must be prefixed with "USINT#".

#### **2) Status Values Returned by Removable Media Function Blocks**

##### **Value Description**

- 0     Operation completed successfully
- 1    End of file was reached before completing
- 2    Function is active, waiting for operation to complete
- 3    Function is waiting on another RM function to complete
- 4    Function block is inactive (usually no power flow)
  
- 1     Card present but unknown format
- 2     No card in slot
- 3     Card present, but not supported
- 4     Card swapped before operation was complete
- 5     Unknown error
  
- 66    File / Path specified does not exist
- 73    Bad file descriptor (corrupt file)
- 77    Attempt to open / rename file that is open
- 81    Specified file already exist
- 86    Function block contains illegal parameter
- 88    Too many open files\*
- 92    Attempt to write failed
- 94    Sharing violation\*
- 95    No disk present\*
- 96    Directory structure corrupt
- 98    Incorrect data format

### ***ST Language***

(\* RD is a declared instance of READ\_RM2 function block \*)

```
RD(EN, #TYPE, FILENAME, OFFSET, NUM, DEST[], #FILETYPE);
STATUS := RD.Status;
```

### ***FBD Language***

### ***LD Language***

### ***IL Language***

(\* RD is a declared instance of READ\_RM2 function block \*)

```
OP1 : CAL RD(EN, #TYPE, FILENAME, OFFSET, NUM, DEST[], #FILETYPE)
      LD RD.Status
      ST Status
```

### ***See also***

[READ\\_RM](#) [READ\\_RM2](#) [WRITE\\_RM](#) [WRITE\\_RM2](#) [RENAME\\_RM](#)  
[DELETE\\_RM](#)

## **Write\_RM**

*Operator* – Perform writing a comma separated value onto a fixed File name on the Removable Media interface.

### **Inputs**

EN : Enable input (TYPE : BOOL)

#MODE: (TYPE : DINT)

This is the writing mode for the function.

- Create - create a new file, error if file DOES exist
- Append - add data to end of existing file, error if file does NOT exist
- Create / Append - create the file if it doesn't exist, append if the file does exist
- Overwrite - if the file exists overwrite with a new file

#TYPE: (TYPE : DINT)

This is the type of data that is written. There is no type or size information encoded in a CSV file and it is the programmer's responsibility to write data to a file using the correct type.

FILENAME: (TYPE : STRING)

This is the fixed filename to write the values from the controller.

SRC[ ] : (TYPE : ANY i.e. DINT[], REAL[], UDINT[], ASCII[] types)

This is a controller register where the data to write is located. Because each element can require more than one 16-bit registers (DINT, REAL, UDINT, ASCII types) and more than one element can be written at a time this can require a large number of registers from this starting point.

NUM: (TYPE : INT)

This determines the number of element to write it can be a constant or 16-bit controller register.

COL/ROW: (TYPE : INT)

This defines the format for writing data to the CSV file.

#FILE TYPE: (TYPE : DINT)

This defines the extension of the file name.

#END OF ROW: (TYPE : BOOL)

Setting this option will cause the row to end at the end of this write function.

### **Outputs**

STATUS : (TYPE : DINT)

This is a 32-bit controller register used to show the status of the function block. It indicates the operation's success/failure code. The first 16-bit register is a status code; see the possible status codes in the status section below. The second 16-bit register shows the number of elements successfully read.

### **Remarks**

#### **1) FILENAME**

This is a constant. It can be up to 147 characters long that includes sub directories (i.e. "my\_data\test.csv").

File and directory names are limited to the old DOS 8.3 convention. This is 8 characters for the name and 3 characters for an extension with a period (.) separating them. See the Removable Media Filenames section for more filename options.

## **2 ) COL/ROW**

This can be a constant. When a CSV file is written to a table format it can be viewed in a column / row format like a spreadsheet. Setting this parameter determines the number of elements to write in a row before a new row is started.

Setting this value to zero will disable the generation of new rows and will generate all data as a single row.

Examples:

3 columns per row

1	2	3
4	5	6
7	8	9

5 columns per row

1	2	3	4	5
6	7	8	9	10

## **3) Status Values Returned by Removable Media Function Blocks**

### **ValueDescription**

- |    |  |
|----|--|
| 0  | Operation completed successfully                       |
| -1 | End of file was reached before completing              |
| -2 | Function is active, waiting for operation to complete  |
| -3 | Function is waiting on another RM function to complete |
| -4 | Function block is inactive (usually no power flow)     |
|    |  |
| 1  | Card present but unknown format                        |
| 2  | No card in slot  |
| 3  | Card present, but not supported                        |
| 4  | Card swapped before operation was complete             |
| 5  | Unknown error  |
|    |  |
| 66 | File / Path specified does not exist                   |
| 73 | Bad file descriptor (corrupt file)                     |
| 77 | Attempt to open / rename file that is open             |
| 81 | Specified file already exist                           |
| 86 | Function block contains illegal parameter              |
| 88 | Too many open files*                                   |
| 92 | Attempt to write failed                                |
| 94 | Sharing violation*                                     |
| 95 | No disk present*                                       |

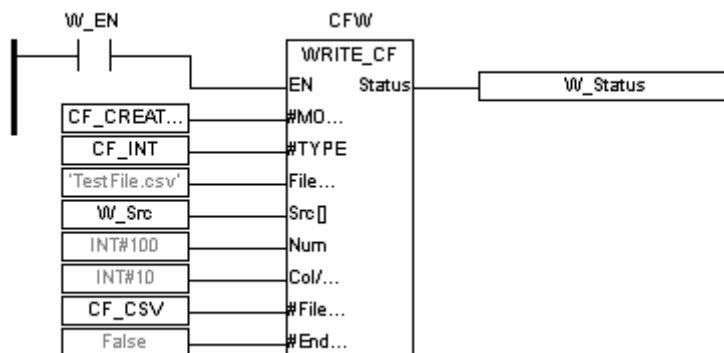
- 96 Directory structure corrupt
- 98 Incorrect data format

### **ST Language**

(\* WR is a declared instance of WRITE\_RM function block \*)

```
WR(EN, #MODE, #TYPE, FILENAME, SRC[], NUM, COL/ROW, # FILETYPE, #
ENDOFROW);
STATUS := WR.Status;
```

### **FBD Language**



### **LD Language**

### **IL Language**

(\* WR is a declared instance of WRITE\_RM function block \*)

```
OP1: CAL WR(EN, #MODE, #TYPE, FILENAME, SRC[], NUM, COL/ROW,
# FILETYPE, # ENDOFROW)
LD WR.Status
ST Status
```

### **See also**

READ\_RM READ\_RM2 WRITE\_RM WRITE\_RM2 RENAME\_RM  
DELETE\_RM

## **Write\_RM2**

*Operator* – Perform writing a comma separated value file to the Removable Media interface where the filename is to be obtained from OCS registers.

### **Inputs**

EN : Enable input (TYPE : BOOL)

#MODE: (TYPE : DINT)

This is the writing mode for the function.

- Create - create a new file, error if file DOES exist
- Append - add data to end of existing file, error if file does NOT exist
- Create / Append - create the file if it doesn't exist, append if the file does exist
- Overwrite - if the file exists overwrite with a new file

#TYPE: (TYPE : DINT)

This is the type of data that is written. There is no type or size information encoded in a CSV file and it is the programmer's responsibility to write data to a file using the correct type.

FILENAME: (TYPE : USINT)

This is the filename to write obtained from OCS registers.

SRC[ ] : (TYPE : ANY i.e. DINT[], REAL[], UDINT[], ASCII[] types)

This is a controller register where the data to write is located. Because each element can require more than one 16-bit registers (DINT, REAL, UDINT, ASCII types) and more than one element can be written at a time this can require a large number of registers from this starting point.

NUM: (TYPE : INT)

This determines the number of element to write it can be a constant or 16-bit controller register.

COL/ROW: (TYPE : INT)

This defines the format for writing data to the CSV file.

#FILE TYPE: (TYPE : DINT)

This defines the extension of the file name.

#END OF ROW: (TYPE : BOOL)

Setting this option will cause the row to end at the end of this write function.

### **Outputs**

STATUS : (TYPE : DINT)

This is a 32-bit controller registers used to show the status of the function block. The first 16-bit register is a status code; see the possible status codes in the status section below. The second 16-bit register shows the number of elements successfully read.

### **Rename**

#### **1) FILENAME**



The filename is stored in contiguous block of OCS registers and has a limit of 147 characters and must be terminated with a NULL (byte containing zero). A percent (%) symbol before the register name is required indicate the file is in a register. This shows that "%R1234" is a valid register reference.

File and directory names are limited to the old DOS 8.3 convention. This is 8 characters for the name and 3 characters for an extension with a period (.) separating them. See the Removable Media Filenames section for more filename options.

USINT - Unsigned Short Integer: An 8-bit unsigned value. Unsigned Short Integers are used where the value of the data is **expected to be in the range of 0 (zero) to 255**.

## 2 ) COL/ROW

This can be a constant. When a CSV file is written to a table format it can be viewed in a column / row format like a spreadsheet. Setting this parameter determines the number of elements to write in a row before a new row is started.

## 3) Status Values Returned by Removable Media Function Blocks

### ValueDescription

0	Operation completed successfully
-1	End of file was reached before completing
-2	Function is active, waiting for operation to complete
-3	Function is waiting on another RM function to complete
-4	Function block is inactive (usually no power flow)
1	Card present but unknown format
2	No card in slot
3	Card present, but not supported
4	Card swapped before operation was complete
5	Unknown error
66	File / Path specified does not exist
73	Bad file descriptor (corrupt file)
77	Attempt to open / rename file that is open
81	Specified file already exist
86	Function block contains illegal parameter
88	Too many open files*
92	Attempt to write failed
94	Sharing violation*
95	No disk present*
96	Directory structure corrupt
98	Incorrect data format

### ST Language

(\* WD is a declared instance of WRITE\_RM2 function block \*)

```
WD(EN, #MODE, #TYPE, FILENAME, SRC[], NUM, COL/ROW, #FILETYPE,  
#ENDOFROW);  
STATUS := WD.Status;
```

### ***FBD Language***

### ***LD Language***

### ***IL Language***

```
(* WD is a declared instance of WRITE_RM2 function block *)  
OP1: CAL WD(EN, #MODE, #TYPE, FILENAME, SR[]C, NUM, COL/ROW,  
#FILETYPE, #ENDOFROW)  
    LD WD.Status  
    ST Status
```

### ***See also***

READ\_RM READ\_RM2 WRITE\_RM WRITE\_RM2 RENAME\_RM  
DELETE\_RM

## **Copy\_CF**

*Operator* – Performs copying of a file on the Removable Media card. The data in the file is not changed. #

### **Inputs**

**EN** : The rung state in a LD diagram is always Boolean. Blocks are connected to the rung with their input and output. The "EN" input is a trigger condition The block is executed only if EN is TRUE (TYPE : BOOL)

**Source:** (TYPE: String)

This is the name of the file to be copied.

**Dest:** (TYPE: String)

This is the file which will be created containing the copied data

### **Outputs**

**STATUS** : (TYPE : DINT)

This is a 16-bit controller registers used to show the status of the function block. See the possible status codes in the status section below.

### **Remarks**

#### **1) Source & Dest**

The filename may be either fixed or obtained by reading from a contiguous block of registers in the *i<sup>3</sup>*, and may includes sub directories (i.e. "my\_data\test.csv"), Where a fixed string is used, it may be up to 147 characters long. If the filename is obtained from *i<sup>3</sup>* registers it is still limited to 147 characters and must be terminated with a NULL (byte containing zero).

If the filename is to come from the *i<sup>3</sup>* and registers and is entered as a register tag ensure that a percent (%) symbol appears before the register name. This is used to differentiate between "R1234" which is a valid file name and "%R1234" which is a register reference.

File and directory names are limited to the old DOS 8.3 convention. This is 8 characters for the name and 3 characters for an extension with a period (.) separating them. See the Removable Media Filenames section for more filename options.

#### **2 ) Status Values Returned by Removable Media Function Blocks**

##### **Value Description**

- |    |  |
|----|--|
| 0  | Operation completed successfully                       |
| -1 | End of file was reached before completing              |
| -2 | Function is active, waiting for operation to complete  |
| -3 | Function is waiting on another RM function to complete |
| -4 | Function block is inactive (usually no power flow)     |
|    |  |
| 1  | Card present but unknown format                        |
| 2  | No card in slot  |

- 3 Card present, but not supported
- 4 Card swapped before operation was complete
- 5 Unknown error

- 66 File / Path specified does not exist
- 73 Bad file descriptor (corrupt file)
- 77 Attempt to open / rename file that is open
- 81 Specified file already exist
- 86 Function block contains illegal parameter
- 88 Too many open files\*
- 92 Attempt to write failed
- 94 Sharing violation\*
- 95 No disk present\*
- 96 Directory structure corrupt
- 98 Incorrect data format

### 3) System Registers used with Removable Media

**%SR175 Status** - This shows the current status of the Removable Media interface.  
Possible status values:

- 0 Removable Media Interface OK
- 1 Card Present but unknown format
- 2 No card in slot
- 3 Card present, but not supported
- 4 Card swapped before operation was complete
- 5 Unknown error

**%SR176 Free Space** - This 32-bit register shows the free space on the Removable Media card in bytes.

**%SR178 Card Capacity** - This 32-bit register shows the total card capacity in bytes.

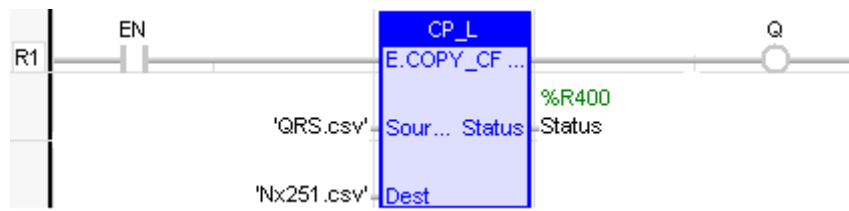
#### ST Language

```
CP (CP_EN,'Testfile.csv','IMO.csv');
Status := CP.Status;
```

#### FBD Language



#### LD Language



**EN/Rung Power** - This function passes power once the Status returns a 0, assuming power is still applied to the function. Should power be lost to the Copy function before it is finished, however, the function will still complete. The block is executed only if EN is TRUE. The output always represents the same status as the "EN" input. Data type is BOOL.

#### IL Language

BEGIN\_IL

cal C (C\_Enable, 'JP.CSV', 'IMO.csv' )

ld C.Status

st C\_Status

END\_IL

#### See also

READ\_RM READ\_RM2 WRITE\_RM WRITE\_RM2 RENAME\_RM  
DELETE\_RM

## Counter Operations

### *Counter Operations*

Below are the standard blocks for managing counters:

Up Counter  
Down Counter

Up counter  
Down counter

## CTD

*Function* - Down counter.

### Inputs

CD : BOOL Enable counting. Counter is decreased on each call when CD is TRUE

LOAD : BOOL Re-load command. Counter is set to PV when called with LOAD to TRUE

PV : DINT Programmed maximum value

### Outputs

Q : BOOL TRUE when counter is empty, i.e. when CV = 0

CV : DINT Current value of the counter

### Remarks

The counter is empty (CV = 0) when the application starts. Counter is set to PV when called with LOAD to TRUE. The counter does not include a pulse detection for CD input. Use R\_TRIG or F\_TRIG function block for counting pulses of CD input signal. In LD language, CD is the input rung. The output rung is the Q output.

### ST Language

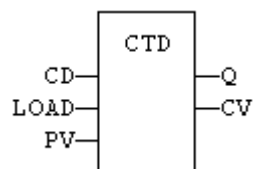
(\* MyCounter is a declared instance of CTD function block \*)

MyCounter (CD, LOAD, PV);

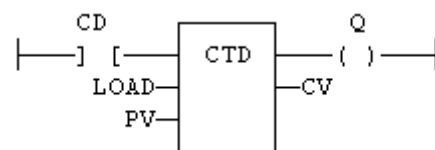
Q := MyCounter.Q;

CV := MyCounter.CV;

### FBD Language



### LD Language



### IL Language

(\* MyCounter is a declared instance of CTD function block \*)

Op1: CAL MyCounter (CD, LOAD, PV)

LD MyCounter.Q

ST Q

LD MyCounter.CV

ST CV

### See also

CTU

CTU

*Function* - Up counter.

### **Inputs**

CU : BOOL Enable counting. Counter is increased on each call when CU is TRUE

RESET : BOOL Reset command. Counter is reset to 0 when called with RESET to TRUE

PV : DINT Programmed maximum value

### **Outputs**

Q : BOOL TRUE when counter is full, i.e. when  $CV = PV$

CV : DINT Current value of the counter

### **Remarks**

The counter is empty ( $CV = 0$ ) when the application starts. The counter does not include a pulse detection for CU input. Use R\_TRIG or F\_TRIG function block for counting pulses of CU input signal. In LD language, CU is the input rung. The output rung is the Q output.

### **ST Language**

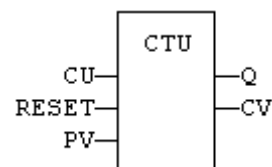
(\* MyCounter is a declared instance of CTU function block \*)

MyCounter (CU, RESET, PV);

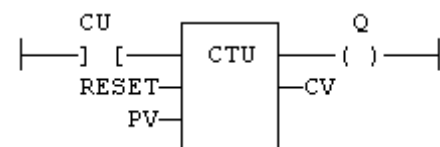
Q := MyCounter.Q;

CV := MyCounter.CV;

### **FBD Language**



### **LD Language**



### **IL Language**

(\* MyCounter is a declared instance of CTU function block \*)

Op1: CAL MyCounter (CU, RESET, PV)

LD MyCounter.Q

ST Q

LD MyCounter.CV

ST CV

### **See also**

CTD



## **Time and Date Operations**

### ***Time and Date Operations***

Below are the standard functions for managing Time and Date operations:

Days of Month

Days of Week

Months of Year

Start and End Year

Time of Day

## Time and Date Operations

Days of Month | Days of Week | Months of Year | Start and End Year | Time of Day

### Configuring Elements

To configure the element, Time and Date Operations assign either fixed or Register Type and Offset (address) as input. The fixed or register values are compared with the current RTC time in the i<sup>3</sup>.

### Power Flow through the Element

When the current RTC time / day / date / month / year of i<sup>3</sup> is between the input given, power is passed through the element to its output, which can be used to set or clear an indicator coil. For example:

Element	Inputs Given	Power Flow
Time of Day	Start time: 11:45 End time : 16:45	The power will be passed when the RTC time in the i <sup>3</sup> is between 11:45 and 16:45 everyday.
Days of Week	Monday, Wednesday and Friday	The power will be passed on Mondays, Wednesdays and Fridays of every week.
Days of Month	5, 15 and 25	The power will be passed on 5 <sup>th</sup> , 15 <sup>th</sup> and 25 <sup>th</sup> of every month.
Months of Year	January, July	The power will be passed from 1 <sup>st</sup> to 31 <sup>st</sup> January and 1 <sup>st</sup> to 31 <sup>st</sup> July of every year.
Start and End Year	Start Year: 2007 End Year: 2009	The power will be passed from year 2007 to 2009.

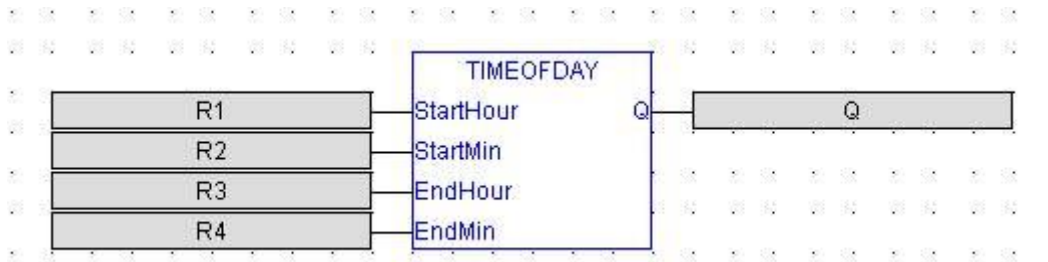
### Time of Day

The Time of Day element compares the current time with the start and end time values. If the current time is between the start time and end time, power is passed. The inputs can be fixed or can be entered using address registers.

### STLanguage

```
Q5:=TimeOfDay( StartHour(*INT*), StartMin(*INT*), EndHour(*INT*),  
EndMin(*INT*) );
```

### FBD Language



### LD Language



### IL Language

```
LD StartHour
TimeOfDay StartMin, EndHour, EndMin
ST Q5
```

### Days of Week

The Days of week element compares the current day of the week with the input and passes power when the current day is one of the inputs.

The input can be fixed or Register Type and Offset. In case of fixed input, the values represent each day from Sunday (1) to Saturday (7). In case of register type and offset, a 16 bit register is used for representing all days with each bit of the register representing one day

i.e. BIT 1 - Sunday  
BIT 7 – Saturday

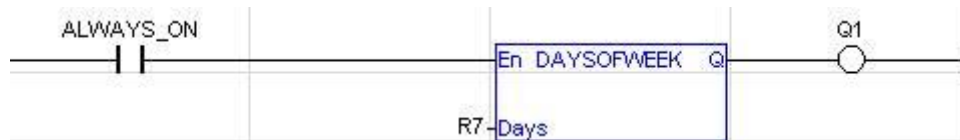
### ST Language

```
Q4:= DaysOfWeek( Days1 ); (*INT*)
```

### FBD Language



### LD Language



### IL Language

LD Week (\* load parameter\*)  
 DaysOfWeek (\*load block\*)  
 ST Q2 (\*store to Q1\*)

### Days of Month

The Days of month element compares the current date of the month with the input and passes power when the current date is one of the inputs.

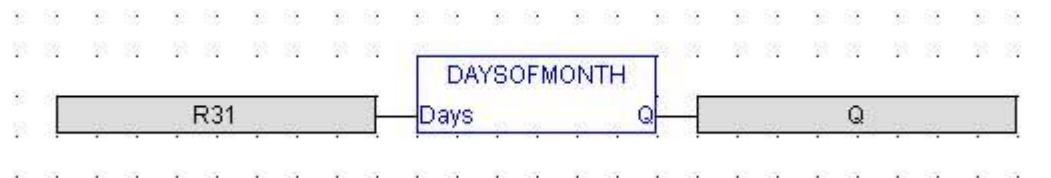
The input can be fixed or Register Type and Offset. In case of fixed input, the values represent each date from 1 to 31st. In case of register type and offset, a 32 bit register is used for representing all dates with each bit of the register representing one date.

i.e. BIT 1 - 1<sup>st</sup> of the month  
 BIT 31 - 31<sup>st</sup> of the month

### ST Language

Q1:= DaysOfMonth( Days );(\*UDINT\*)

### FBD Language



### LD Language



## IL Language

LD Days (\* load parameter\*)  
DaysOfMonth (\*load block\*)  
ST Q1 (\*store to Q1\*)

## Months of Year

The Months of Year element compares the current month of the year with the input and passes power when the current month is one of the inputs.

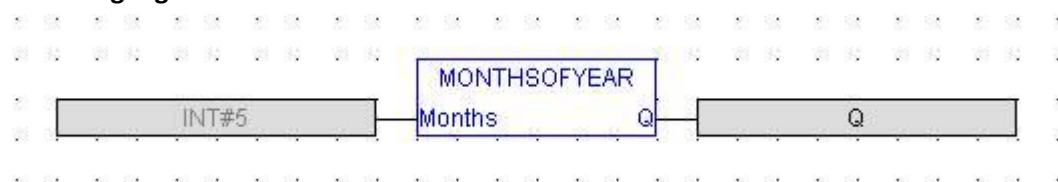
The input can be fixed or Register Type and Offset. In case of fixed input, each value represents a month from January (1) to December (12). In case of register type and offset, a 16 bit register is used for representing all months with each bit of the register representing one month.

i.e. BIT 1 - January  
BIT 12 – December

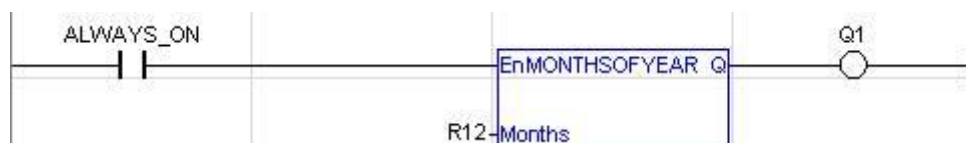
## ST Language

Q2:= MonthsOfYear( Months); (\*INT\*)

## FBD Language



## LD Language



## IL Language

LD Months (\* load parameter\*)  
MonthsOfYear (\*load block\*)  
ST Q3 (\*store to Q1\*)

## Start and End Year

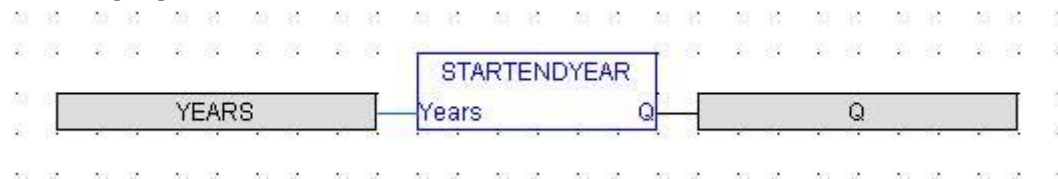
The Start and End Year element compares the current year with the inputs and passes power when the current year is between the inputs.

The input can be constants (fixed) or register type and offsets. In case of fixed input, start and end values can be directly entered. In case of register type and offset, a 32 bit register is used for specifying start and end year. The higher 16 bits represent end year and the lower 16 bits represent the start year.

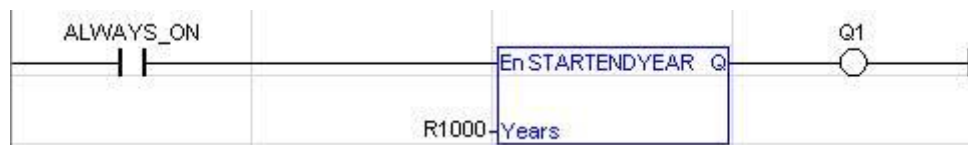
### ST Language

```
Q3:= StartEndYear( Years ); (*DINT*)
```

### FBD Language



### LD Language



### IL Language

```
LD Year (* load parameter*)
StartEndYear (*load block*)
ST Q4 (*store to Q1*)
```

## Move Operations

### ***Move Operations***

Below are the standard operators that perform Move Operations:

Fill Block	Fills a source register values to destination location
Multiple Shift	Shifts array of Elements to left or right a variable No. of elements
Multiple Rotation	Rotates array of Elements to left or right a variable No. of elements
Move Data	Moves a block of register values from source to destination
Binary Selector	Select one of the inputs - 2 inputs

## **FILL**

*Operator* – This element fills a source register values from src location to destination location.

**NOTE:** The **Fill** element operates on 16-bit data *only*.

**WARNING:** If the **IN** value is a *signed* numeric constant, it is treated as an unsigned number when the element is configured. For example, if **IN** is configured as '**-1**', the value '**65535**' is used.

### **Inputs**

**SRC** – (TYPE : INT)

Source value can be either an integer constant or the value contained in another register. For e.g. R1 is containing 123.

**DST[ ]** - (TYPE : Any[])

This is output starting register. For e.g. R10 and count is 5 then start to fill the value as follow:

R10 – 123

R11 - 123

R12 – 123

R13 - 123

and R14 value is 123.

**#COUNT** – (TYPE : INT)

It should be a constant value and enter in INT#5 format.

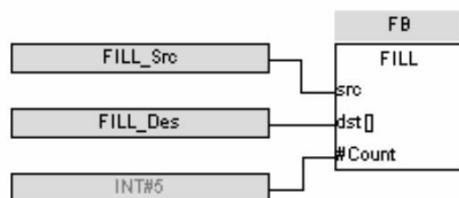
For e.g. Count value is 5.

### **ST Language**

(\* FILL1 is a declared instance of FILL function block \*)

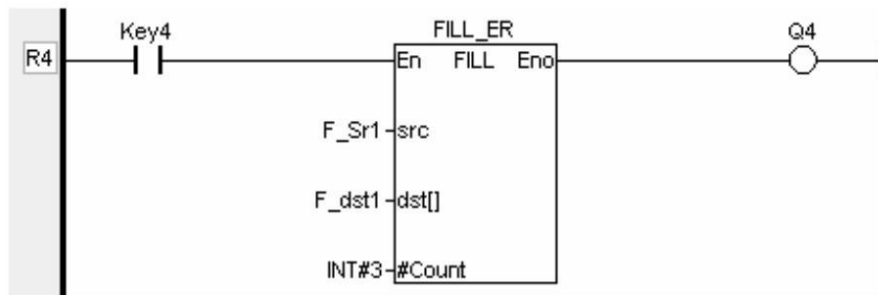
FILL1(SRC, DST[], #COUNT);

### **FBD Language**



### **LD Language**





### IL Language

(\* FILL1 is a declared instance of FILL function block \*)

Op1: CAL FILL1(SRC, DST[], #COUNT)

### See also

[MultiRotate](#) [MultiShift](#) [MVB](#)

## MultiRot

**Operator** – This function allows an array of BITS, BYTES, WORDS, and DWORDS to be rotated left or right a variable numbers of elements.

### Inputs

#### Power Flow – (TYPE : BOOL)

When the input to this function block is high it completes a rotate as specified by the parameters every scan. This function is not edge sensitive. This function always passes power flow.

#### N – (TYPE : INT)

This is the number of elements to rotate. This can be a constant or a WORD variable. The N (number) range is less than or equal to the LEN value.

#### SRC[ ] – (TYPE : ANY[])

This is the starting BIT, BYTE, WORD or DWORD for the array to be rotated. After the data is rotated it is stored in the array of data starting at this location. BIT arrays can start at any location (%I1, %I6, %R1.1, and % R4.7...). BYTE, WORD, and DWORD arrays must start on a WORD boundary (%I1, %I17, %I33, %R1, and % R2...).

#### #LEN – (TYPE : DINT)

This is the number of BITS, BYTES, WORDS, or DWORDS in the array. This must be a constant number from 1 to 32767.

BIT: 1 to 32767

BYTE: 1 to 4096

WORD: 1 to 2048

DWORD: 1 to 1024

**Left** - This is the direction to rotate. If this input is high the data is rotated to the left. If this input is low the data is rotated to the right.

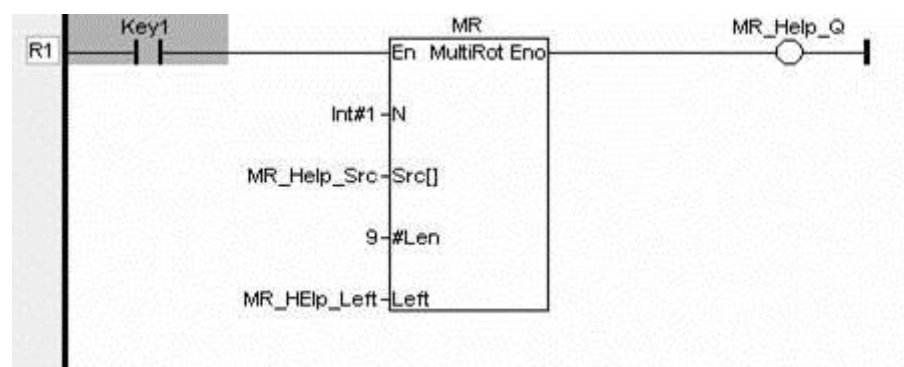
### ST Language

(\* MR is a declared instance of MultiRot function block \*)

```
MR(N, SRC[], #LEN, LEFT);
```

### FBD Language

### LD Language



En is the Enable input & Eno is the enable output. En & Eno will be the same state. The other functionality is similar to that of a FBD.

### ***IL Language***

(\* MR is a declared instance of MultiRot function block \*)

CAL MR(N, SRC[], #LEN, LEFT)

### ***See also***

MultiShift MVB FILL

## MultiShift

**Operator** – This function allows an array of BITS, BYTES, WORDS, and DWORDS to be shifted left or right a variable numbers of elements.

### Inputs

#### Power Flow – (TYPE : BOOL)

When the input to this function block is high it completes a shift as specified by the parameters every scan. This function is not edge sensitive. This function always passes power flow.

#### N - (TYPE : INT)

This is the number of elements to shift. This can be a constant or a WORD variable. The N (number) range is less than or equal to the LEN value.

#### SRC[ ] – (TYPE : ANY[])

This is the starting BIT, BYTE, WORD or DWORD for the array to be shifted. After the data is shifted it is stored in the array of data starting at this location. BIT arrays can start at any location (%I1, %I6, %R1.1, and % R4.7...). BYTE, WORD, and DWORD arrays must start on a WORD boundary (%I1, %I17, %I33, %R1, and % R2...).

#### #LEN – (TYPE : DINT)

This is the number of BITS, BYTES, WORDS, or DWORDS in the array. This must be a constant number as follows:

BIT: 1 to 32767

BYTE: 1 to 4096

WORD: 1 to 2048

DWORD: 1 to 1024

#### Left - (TYPE : BOOL)

This is the direction to shift. If this input is high the data is shifted to the left. If this input is low the data is shifted to the right.

#### IN - (TYPE : ANY)

This is the BIT, BYTE, WORD, or DWORD to shift into the array.

#### @OUT - (TYPE : ANY)

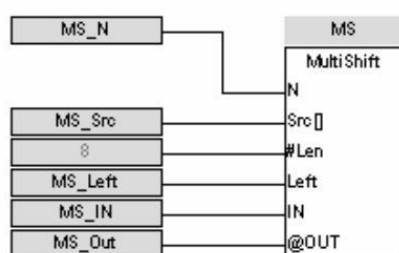
This is the last BIT, BYTE, WORD or DWORD shifted out of the array.

### ST Language

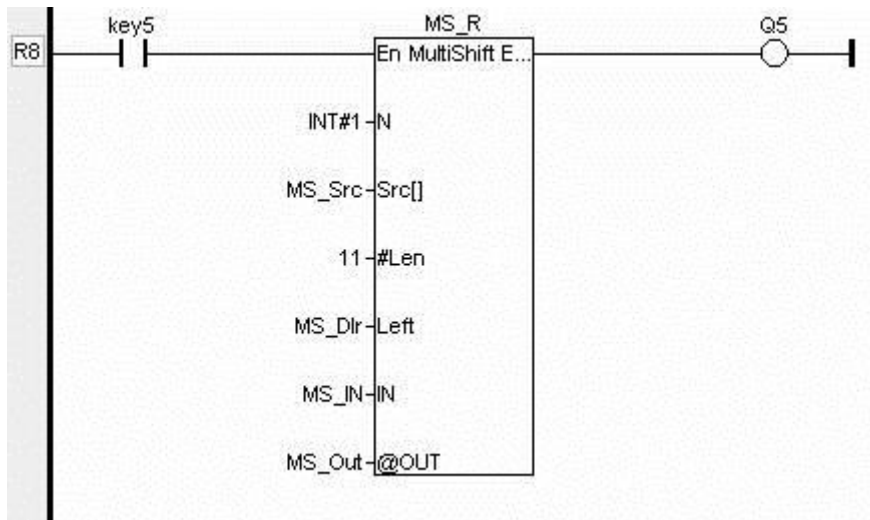
(\* MS is a declared instance of MultiShift function block \*)

MS(N, SRC[], #LEN, LEFT, @OUT, IN);

### FBD Language



### LD Language



En is the Enable input & Eno is the enable output. En & Eno will be the same state. The other functionality is similar to that of a FBD.

### ***IL Language***

(\* MS is a declared instance of MultiShift function block \*)  
 CAL MS(N, SRC[], #LEN, LEFT, IN, @OUT)

### ***See also***

MultiRot MVB FILL

## Move Block

**Operator** – This element moves a block of register values from source to destination location.

### Inputs

**SRC[ ] – (TYPE : ANY[])**

Source value can be either an integer constant or the value contained in another register and assigns the dimension of register.

For e.g. Sr[0] to Sr[4] is containing 28.

**DST[ ] – (TYPE : ANY[])**

This is output register and assign with dimension.

For e.g. Dest[0] to Dest[4] and count is 5 then it moves from Sr0 – Sr4 value to Dest0 – Dest4.

**# COUNT – (TYPE : INT)**

It should be a constant value and enter in INT#5 format.

For e.g. Count value is 5.

### ST Language

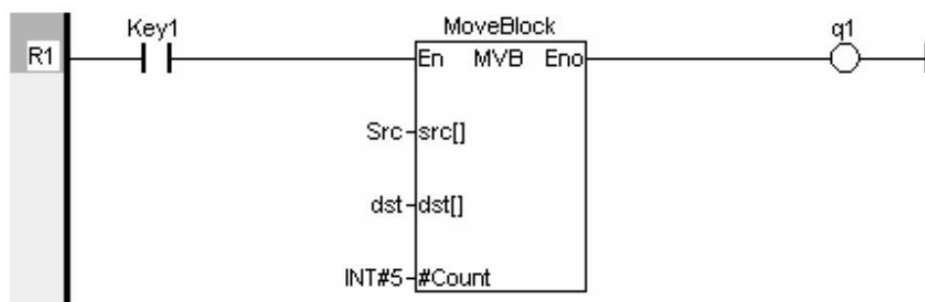
(\* MVB1 is a declared instance of MVB function block \*)

MVB1(SRC[], DST[], #COUNT);

### FBD Language



### LD Language



### IL Language

(\* Move is a declared instance of MVB function block \*)  
Op1: CAL Move(SRC[], DST[], #COUNT)

**See also**

[MultiRotate](#) [MultiShift](#) [FILL](#)

## SEL

*Function* - Select one of the inputs - 2 inputs.

### Inputs

SELECT : BOOL Selection command

IN1 : DINT First input

IN2 : DINT Second input

### Outputs

Q : DINT IN1 if SELECT is FALSE; IN2 if SELECT is TRUE

### Truth table

SELECT	Q
0	IN1
1	IN2

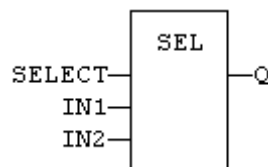
### Remarks

In LD language, the selector command is the input rung. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by comas.

### ST Language

Q := SEL (SELECT, IN1, IN2);

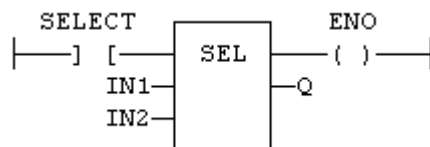
### FBD Language



### LD Language

(\* the input rung is the selector \*)

(\* ENO has the same value as SELECT \*)



### IL Language

Op1: LD SELECT

SEL IN1, IN2

ST Q



## **PID Operations**

### ***PID Operations***

Below are the standard blocks for PID operations:

Get PID Manual Mode

Status                      PID independent

Independent PID Loop    PID independent with Auto Tuning feature

Independent PID with    PID ISA

Auto Tune                PID ISA with Auto Tuning feature

ISA PID Loop             Sets PID registers

ISA PID with Auto

Tune

Set PID Control Block

Set PID MAN

## GETPIDMAN

Operator: Gets the Control Output value of the PID associated with the Control block given as Input and loads it in the Output Register.

### Inputs

CB[ ] : Input the values of the register usage mentioned below (TYPE : INT[])

### Output

Q: The control variable result of the associated PID block.

Enter a register address, or select a named register. This is the location of the Control Variable value of the associated PID Loop going out to the process. This value may NOT be a decimal constant.

### Remarks

Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the reference Array.

Registers at offset 0 through 9 must be configured before the PID element is used.

This is configured using the SETPID block.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect:

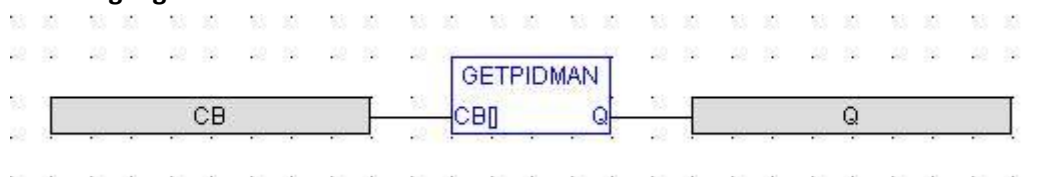
				$K_d * \text{delta Error} / dt.$
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt.$
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

#### ST Language

```
Q:= GETPIDMAN( CB(*INT*) );
```

#### FBD Language



#### Ladder Language



#### IL Language

```
GETPIDMAN(CB1)
ST Q1
```

**See Also:**

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## PID\_IND

Operator - Performs the proportional integral derivative (PID) algorithm.

Inputs

CB[ ] : Input the values of the register usage mentioned below. (TYPE : INT[])

SP : Process Setpoint (TYPE : INT)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may NOT be a decimal constant.

PV : Process Variable (TYPE : INT)

Enter a register address or select a named register. This is the location (typically %AI) of the Process Variable value coming in from the process. This value may NOT be a decimal constant.

MAN EN : Manual / Auto Boolean Switch (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register, typically %T.

UP : Manual Mode up adjustment input (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register, typically %T.

DOWN : Manual Mode down adjustment (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register, typically %T.

Outputs

CV : The control variable result (TYPE : INT)

Enter a register address, or select a named register. This is the location (typically %AQ) of the Control Variable value going out to the process. This value may NOT be a decimal constant.

## Remarks

Independent PID:

$$CV_{out} = (K_p * Error) + (K_i * Error * dt) + (K_d * Derivative) + CV_{Bias}$$

Where:

$dt$  = Internal elapsed time clock - previous elapsed time clock

$Derivative = (Error - previous\ Error) / dt$

--or--

$Derivative = (pv - previous\ PV) / dt$

[User selectable during configuration].



$T_i$  = Integral time

$T_d$  = Derivative time

## Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.

1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>
11	Manual Command	CV Counts	Tracks CV in Auto mode; sets CV in Manual Mode.	In the Automatic mode this register tracks the CV value. In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.
12	Internal SP	Used by 	N/A	Tracks SP in
13	Internal PV	Used by 	N/A	Tracks PV in

14	Internal CV	Used by <i>i<sup>3</sup></i>	N/A	Tracks CV out
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

Each PID element must use a distinctly separate Reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

Registers at offset 0 through 9 must be configured before the PID element is used.

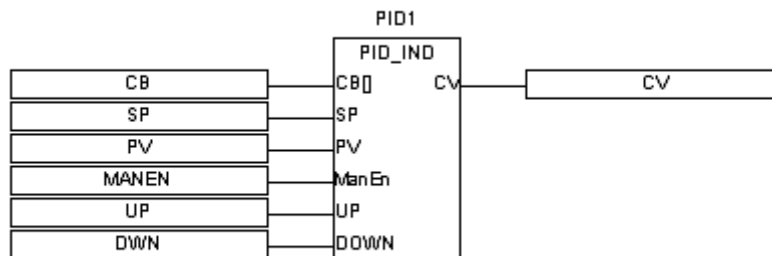
### ST Language

(\* PID1 is a declared instance of PID\_IND function block \*)

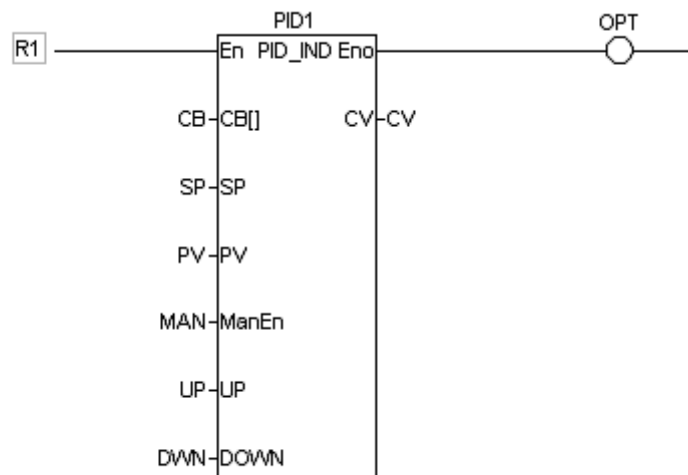
PID1(CB[], SP, PV, MANEN, UP, DOWN);

CV := PID1.CV;

### FBD Language



### LD Language



En is the Enable input & Eno is the enable output. En & Eno will be the same state. The other functionality is similar to that of a FBD.

### IL Language

(\* PID1 is a declared instance of PID\_IND function block \*)

Op1: CAL PID1(CB[], SP, PV, MANEN, UP, DOWN)

LD PID1.CV

ST CV

Caution: Overlapping references will result in erratic operation of the PID algorithm.

***See also***

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)



## PID\_IND\_AUTO

*Operator* - Performs the proportional integral derivative (PID) algorithm with auto tuning function.

### Inputs

**CB[ ]** : Input the values of the register usage mentioned below. (TYPE : INT[])

**SP** : Process Setpoint (TYPE : INT)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may NOT be a decimal constant.

**PV** : Process Variable (TYPE : INT)

Enter a register address or select a named register. This is the location (typically **%AI**) of the Process Variable value coming in from the process. This value may NOT be a decimal constant.

**MAN EN** : Manual / Auto Boolean Switch (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**UP** : Manual Mode up adjustment input (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**DOWN** : Manual Mode down adjustment (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**TUNE** : Input which controls when the function should start the auto tune process. (TYPE : BOOL)

An edge triggered boolean TUNE input starts the autotuning cycle. This input needs to be held high during the autotuning cycle. If it is negated during the AUTOTUNE cycle, the controller stops autotuning and reverts to the previous settings.

**#FILTER** : (TYPE : DINT)

This defines how far above and below the setpoint the process must go when performing the auto tune experiment. Processes with more noise should be setup with a high percentage.

**#RESP** : (TYPE : DINT)

This defines the relative speed of the PID loop once it is tuned.

**#TYPE** : (TYPE : DINT)

This options allows the auto tune procedure to calculate terms for PID, PI or P terms.

**#TUNE2/3** : (TYPE : BOOL)

This allows the auto tuning experiment to change the output based on 2/3 the set point. Use this option when it is not desired for the process to travel above the setpoint during the auto tuning experiment.

### Outputs

**CV** : The control variable result (TYPE : INT)

Enter a register address, or select a named register. This is the location (typically **%AQ**) of the Control Variable value going out to the process. This value may NOT be a decimal constant.

**DONE** : The control variable result (TYPE : BOOL)

This defines an output bit that is set by the function when the auto tune is complete.

### Remarks

**Independent PID\_Auto:**

$$CV_{out} = (K_p * Error) + (K_i * Error * dt) + (K_d * Derivative) + CV_{Bias}$$

Where:

dt = Internal elapsed time clock - previous elapsed time clock

Derivative = (Error - previous Error)/dt

--or--

Derivative = (pv - previous PV)/dt

[User selectable during configuration].

Ti = Integral time

Td = Derivative time

**Register Usage**

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta Error / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * Error * dt$ .

6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>
11	Manual Command	CV Counts	Tracks CV in Auto mode; sets CV in Manual Mode.	In the Automatic mode this register tracks the CV value. In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.
12	Internal SP	Used by <i>i</i> <sup>3</sup>	N/A	Tracks SP in
13	Internal PV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks PV in
14	Internal CV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks CV out
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

Registers at offset 0 through 9 must be configured before the PID element is used.

#### ST Language

(\* PID3 is a declared instance of PID\_IND\_Auto function block \*)

```
PID3(CB[], SP, PV, MANEN, UP, DOWN, TUNE, #FILTER, #RESP, #TYPE, #TUNE2/3);
```

```
CV := PID3.CV ;
```

```
Done :=PID3.Done;
```

#### FBD Language

### ***LD Language***

En is the Enable input & Eno is the enable output. En & Eno will be the same state. The other functionality is similar to that of a FBD.

### ***LT Language***

(\* PID3 is a declared instance of PID\_IND\_Auto function block \*)

Op1: CAL PID3(CB[], SP, PV, MANEN, UP, DOWN, TUNE, #FILTER, #RESP, #TYPE, #TUNE2/3)

LD PID3.CV

ST CV

LD PID3.Done

ST Done

**Caution:** Overlapping references will result in erratic operation of the PID algorithm.

### ***See also***

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## PID\_ISA

*Operator* - Performs the proportional integral derivative (PID) algorithm.

### Inputs

**CB[ ]** : Input the values of the register usage mentioned below. (TYPE : INT[])

**SP** : Process Setpoint (TYPE : INT)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may NOT be a decimal constant.

**PV** : Process Variable (TYPE : INT)

Enter a register address or select a named register. This is the location (typically **%AI**) of the Process Variable value coming in from the process. This value may NOT be a decimal constant.

**MAN EN** : Manual / Auto Boolean Switch (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**UP** : Manual Mode up adjustment input (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**DOWN** : Manual Mode down adjustment (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register, typically **%T**.

### Outputs

**CV** : The control variable result (TYPE : INT)

Enter a register address, or select a named register. This is the location (typically **%AQ**) of the Control Variable value going out to the process. This value may NOT be a decimal constant.

### Remarks

ISA PID :

$CV_{out} = K_p * (Error + (Error * dt / T_i) + (T_d * Derivative)) + CV_{Bias}$

Where:

$dt$  = Internal elapsed time clock - previous elapsed time clock

$Derivative = (Error - previous\ Error) / dt$

--or--

$Derivative = (pv - previous\ PV) / dt$

[User selectable during configuration].

$T_i$  = Integral time

$T_d$  = Derivative time

### Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type **%R**. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.

1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>
11	Manual Command	CV Counts	Tracks CV in Auto mode; sets CV in Manual Mode.	In the Automatic mode this register tracks the CV value. In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.

12	Internal SP	Used by <i>i<sup>3</sup></i>	N/A	Tracks SP in
13	Internal PV	Used by <i>i<sup>3</sup></i>	N/A	Tracks PV in
14	Internal CV	Used by <i>i<sup>3</sup></i>	N/A	Tracks CV out
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

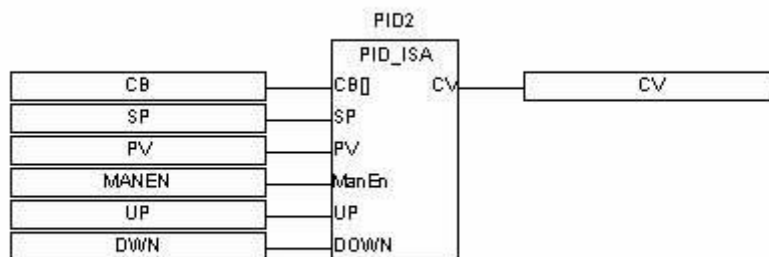
Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

Registers at offset 0 through 9 must be configured before the PID element is used.

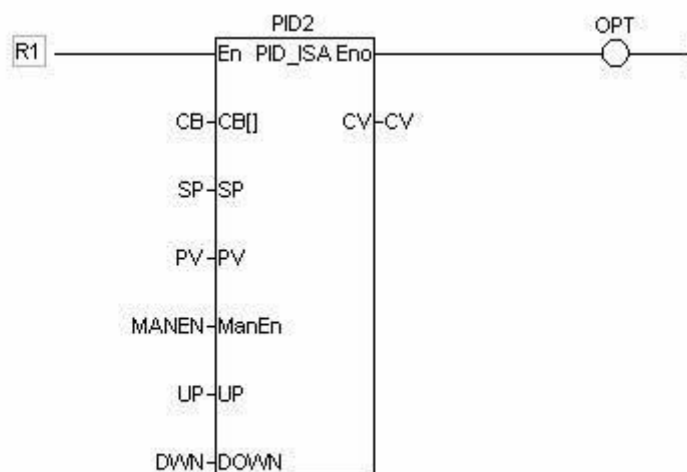
### ST Language

(\* PID2 is a declared instance of PID\_ISA function block \*)  
PID2(CB[], SP, PV, MANEN, UP, DOWN);  
CV := PID2.CV;

### FBD Language



### LD Language



En is the Enable input & Eno is the enable output. En & Eno will be the same state.  
The other functionality is similar to that of a FBD.

### ***IL Language***

(\* PID2 is a declared instance of PID\_ISA function block \*)

```
Op1: CAL PID2(CB[], SP, PV, MANEN, UP, DOWN)
      LD PID2.CV
      ST CV
```

**Caution:** Overlapping references will result in erratic operation of the PID algorithm.

### ***See also***

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)



## PID\_ISA\_AUTO

*Operator* - Performs the proportional integral derivative (PID) ISA algorithm with auto tuning function.

### **Inputs**

**CB[ ]** : Input the values of the register usage mentioned below. (TYPE : INT[])

**SP** : Process Setpoint (TYPE : INT)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may NOT be a decimal constant.

**PV** : Process Variable (TYPE : INT)

Enter a register address or select a named register. This is the location (typically **%AI**) of the Process Variable value coming in from the process. This value may NOT be a decimal constant.

**MAN EN** : Manual / Auto Boolean Switch (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**UP** : Manual Mode up adjustment input (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**DOWN** : Manual Mode down adjustment (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**TUNE** : Input which controls when the function should start the auto tune process. (TYPE : BOOL)

An edge triggered boolean TUNE input starts the autotuning cycle. This input needs to be held high during the autotuning cycle. If it is negated during the AUTOTUNE cycle, the controller stops autotuning and reverts to the previous settings.

**#FILTER** : (TYPE : DINT)

This defines how far above and below the setpoint the process must go when performing the auto tune experiment. Processes with more noise should be setup with a high percentage.

**#RESP** : (TYPE : DINT)

This defines the relative speed of the PID loop once it is tuned.

**#TYPE** : (TYPE : DINT)

This options allows the auto tune procedure to calculate terms for PID, PI or P terms.

**#TUNE2/3** : (TYPE : BOOL)

This allows the auto tuning experiment to change the output based on 2/3 the set point. Use this option when it is not desired for the process to travel above the setpoint during the auto tuning experiment.

## Outputs

**CV** : The control variable result (TYPE : INT)

Enter a register address, or select a named register. This is the location (typically %AQ) of the Control Variable value going out to the process. This value may NOT be a decimal constant.

**DONE** : The control variable result (TYPE : BOOL)

This defines an output bit that is set by the function when the auto tune is complete.

## Remarks

### ISA PID\_Auto :

$$CV_{out} = K_p * (Error + (Error * dt / T_i) + (T_d * Derivative)) + CV_{Bias}$$

Where:

dt = Internal elapsed time clock - previous elapsed time clock

Derivative = (Error - previous Error)/dt

--or--

Derivative = (pv - previous PV)/dt

[User selectable during configuration].

Ti = Integral time

Td = Derivative time

Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta Error / dt$ .

5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>
11	Manual Command	CV Counts	Tracks CV in Auto mode; sets CV in Manual Mode.	In the Automatic mode this register tracks the CV value. In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.
12	Internal SP	Used by <i>i</i> <sup>3</sup>	N/A	Tracks SP in
13	Internal PV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks PV in
14	Internal CV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks CV out
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

Each PID element must use a distinctly separate Reference Array, even if the values are identical to an exiting PID element.

There can be no overlapping of PID elements.

Registers at offset 0 through 9 must be configured before the PID element is used.

### **ST Language**

(\* PID4 is a declared instance of PID\_ISA\_Auto function block \*)

```
PID4(CB[], SP, PV, MANEN, UP, DOWN, TUNE, #FILTER, #RESP, #TYPE, #TUNE2/3);
```

```
CV := PID4.CV;  
DONE := PID4.Done;
```

### ***FBD Language***

### ***LD Language***

En is the Enable input & Eno is the enable output. En & Eno will be the same state.  
The other functionality is similar to that of a FBD.

### ***IL Language***

(\* PID4 is a declared instance of PID\_ISA\_Auto function block \*)

```
Op1:  CAL PID4(CB[], SP, PV, MANEN, UP, DOWN, TUNE, #FILTER, #RESP,  
#TYPE, #TUNE2/3)  
      LD PID4.CV  
      ST CV  
      LD PID4.Done  
      ST DONE
```

**Caution:** Overlapping references will result in erratic operation of the PID algorithm.

### ***See also***

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## SET\_PID

*Operator* - Performs the setting for PID Registers.

### Inputs

**CB[ ]** : Input the values of the register usage mentioned below (TYPE : INT[])

**PERIOD** : Sample Period (TYPE : INT)

**DEADBAND +** : Dead Band + (TYPE : INT)

Defines the Upper Dead Band limits in terms of PV counts.

Set to 0 (zero) if no dead band is required.

**DEADBAND -** : Dead Band - (TYPE : INT)

Defines the Upper Dead Band limits in terms of PV counts.

Set to 0 (zero) if no dead band is required.

Both Deadbands be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.

**Kp** : Proportional Gain (TYPE : INT)

Sets the Proportional Gain (Kp) factor in terms of percent.

100 sets unity gain (gain of 1).

**Kd** : Derivative Gain (TYPE : INT)

Entered as a time with a resolution of 10 mS.

In the PID equation this has the effect:  $K_d * \Delta \text{Error} / dt$ .

**Ki** : Integral Rate (TYPE : INT)

Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect:  $K_i * \text{Error} * dt$ .

**CVBias** : CV Bias (TYPE : INT)

Number of CV counts added to the output before the rate and amplitude clamps.

**CVUpClamp** : CV Upper Clamp (TYPE : INT)

Number of CV Counts that represent the highest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.

**CVDnClamp** : CV Down Clamp (TYPE : INT)

Number of CV Counts that represent the lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.

**MinSlew** : Minimum Slew Time (TYPE : INT)

Determines how fast the CV value can change.

**ErrAction** : Error Action (TYPE : BOOL)

**DAction** : (TYPE : BOOL)

**OutPol** : (TYPE : BOOL)

**Dsense** : (TYPE : BOOL)

### Remarks

Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the reference Array.

Registers at offset 0 through 9 must be configured before the PID element is used.

This is configured using the SetPID block.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments,

				allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

### **ST Language**

(\* SETPID1 is a declared instance of SetPID function block \*)

SetPID1(CB[], Period, DeadBand +, Deadband-, Kp, Kd, Ki, CVBias, CVUpClamp, CVDnClamp, MinSlew, ErrAction, DAction, Outpol, Dsense);

## ***FBD Language***

### ***LD Language***

En is the Enable input & Eno is the enable output. En & Eno will be the same state.  
The other functionality is similar to that of a FBD.

### ***IL Language***

(\* SETPID1 is a declared instance of SetPID function block \*)

Op1: CAL SetPID1(CB[], Period, DeadBand+, Deadband-, Kp, Kd, Ki, CVBias, CVUpClamp, CVDnClamp, MinSlew, ErrAction, DAction, Outpol, Dsense)

Caution: Overlapping references will result in erratic operation of the PID algorithm.

### ***See also***

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## SETPID\_MAN

*Operator:* Sets in Manual mode, the PID associated with the Control block with the Input to this block as Control Output value.

### Inputs

**CB[ ]:** Input the values of the register usage mentioned below (TYPE : INT[])

**IN:** Input value that will be fed to the Control Variable as Manual Output (TYPE: INT)

### Output

**OK:** Block execution indicator, it goes high if the block has been executed successfully.

### Remarks

#### Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the reference Array.

Registers at offset 0 through 9 must be configured before the PID element is used.

This is configured using the SETPID block.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .



5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

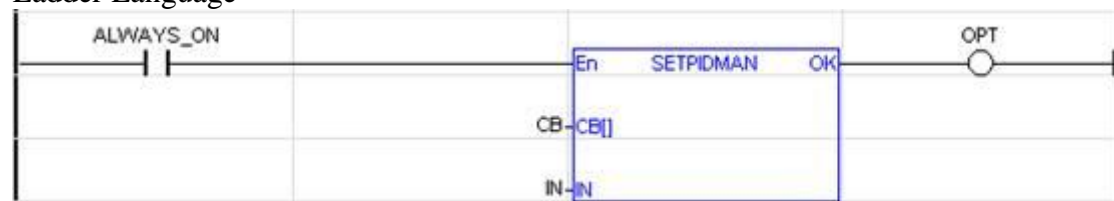
### ST Language

```
SETPIDMAN( CB, IN );
OK := SETPIDMAN. OK;
```

### FBD Language



### Ladder Language



### IL Language

```
OP1: CAL IN;
LD SETPIDMAN(CB)
ST Q
```

See Also:

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## Network Operations

### ***Network Operations***

Below are the standard operators and functions that manage character strings:

Get N Network  
Words

[Put N Network Words](#) Copies global data from any device on the network to into any set of registers

Put Network Heartbeat Sends global data using multiple networks IDs based on SEND trigger input

Get Network Heart Beat Transmit a heartbeat iCAN message indicating it is on-line and operating normally

Put N Network Words Detects network heartbeat from another device

Sends global data using multiple networks IDs

Get Remote Digital IO Receives data from a remote I/O function block and places the received data in a set of registers specified

Put Remote Analog IO Receives data from a remote I/O function block and places the received data in a set of registers specified

Put Remote Digital IO Sending data to a remote I/O function block

Sending data to a remote I/O function block

Get Remote

Analog IO

## Net Get Heartbeat

This function allows the detection of a network heartbeat from another device. This function does not generate any network traffic. This function works only with iCAN networks. This function will not pass power flow if the ID is not in the legal range or if the device being monitored does not send a heartbeat message within the timeout defined by PT.

### Inputs

#### EN – (TYPE: BOOL)

The "EN" input is a condition. If EN is TRUE State then ENO is same.

#### ID – (TYPE: INT)

This is a register or constant defining the ID of the device to monitor for a heartbeat.

#### #PT – (TYPE: INT)

This is the maximum amount of time to wait for the heartbeat from the monitored device. This timeout should be greater than the rate the device is sending heartbeat messages. Depending on network traffic and scan rates the GET timeout should be 10 to 1000 milliseconds greater than the PUT. This has a range of 1 to 6553 milliseconds.

### Outputs

#### Status – (TYPE: INT)

This register is currently used for internal record keeping. Do not allow other functions to write to this register.

#### ENO – (TYPE: BOOL)

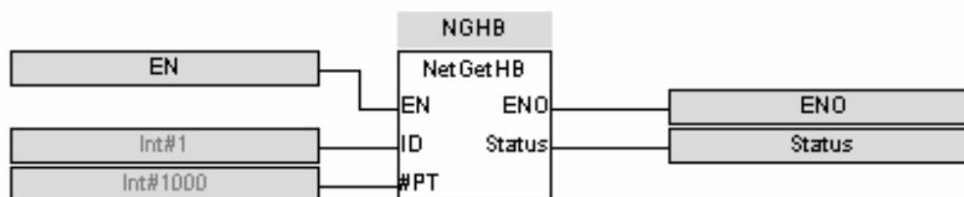
ENO is equal to EN

ST Language

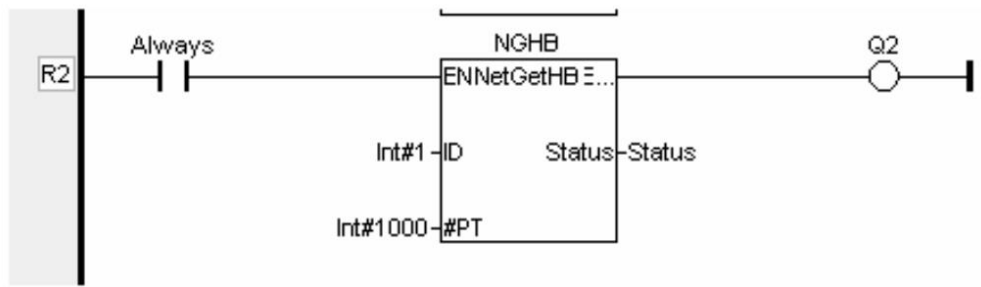
(\* NGHB is a declared instance of NetGetHB function block \*)

```
NGHB(EN, ID, #PT);  
ENO := NGHB.ENO;  
STATUS := NGHB.Status;
```

### FBD Language



### LD Language



IL Language

(\* NGHB is a declared instance of NetGetHB function block \*)

Op1: CAL NGHB(EN, ID, #PT)

LD NGHB.ENO

ST ENO

LD NGHB.STATUS

ST STATUS

**See also**

[NetGetW](#) [NetGetRemoteIO\\_A](#) [NetGetRemoteIO\\_D](#)

### Net Get Remote I/O Function Block (For Analog)

**Operator** - This function handles receiving data from a remote I/O function block and places the received data in a set of registers specified by the user. This function passes power flow if the function is actively receiving data / heartbeat messages from the remote I/O device. This function stops passing power flow if it has not received data / heartbeat messages from the remote I/O device for 2000 milliseconds.

A remote I/O device consists of an iCAN device such as SmartStix modules that transmit global data and receive directed network data.

#### Inputs

##### **ID – (TYPE: INT)**

This is the network ID of the remote I/O from which to receive data. This can be a constant from 1 to 253 or can be a 16-bit register.

##### **DST[ ] – (TYPE: INT[])**

This is the location to start placing received data from the remote I/O device. The number of registers used is defined by the "#N" parameter (see below).

##### **#N – (TYPE: INT)**

This is the number of words to receive from the remote I/O device. Typically for digital devices a 1 to 16 point module requires 1 word of data, a 17-32-point module requires 2 words.

#### Outputs

##### **Status – (TYPE: INT)**

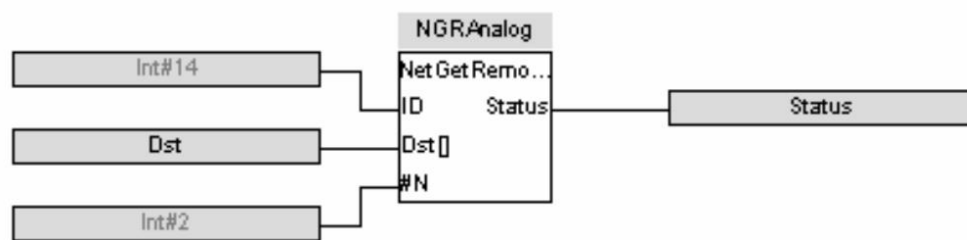
This 16-bit register is used internally. It should be not be written by any other function block. Use the power flow from this function for the pass/fail status.

### ST Language

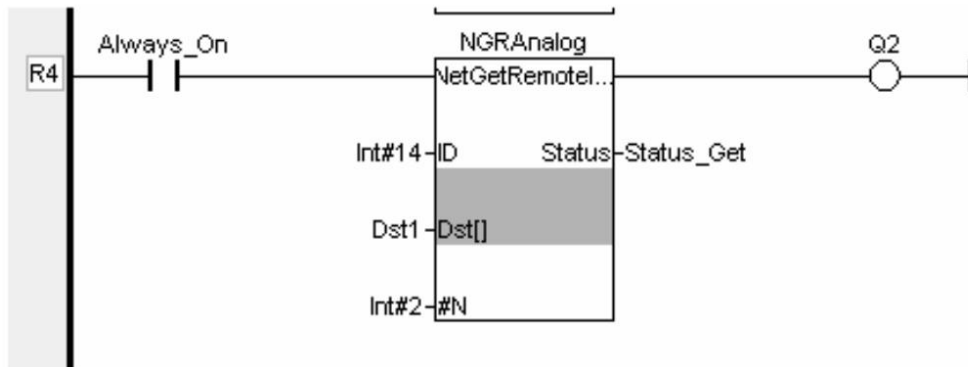
(\* NGRIA is a declared instance of NetGetRemoteIO\_A function block \*)

```
NGRIA(ID, DST[], #N);  
STATUS := NGRIA.Status;
```

### FBD Language



### LD Language



### **IL Language**

(\* NGRIA is a declared instance of NetGetRemoteIO\_A function block \*)

```
Op1: CAL NGRIA(ID, DST[], #N)
      LD NGRIA.Status
      ST STATUS
```

### **See also**

NetGetW NetGetHB NetGetRemoteIO\_D

## Net Get Remote I/O Function Block (For Digital)

*Operator* - This function handles receiving data from a remote I/O function block and places the received data in a set of registers specified by the user. This function passes power flow if the function is actively receiving data / heartbeat messages from the remote I/O device. This function stops passing power flow if it has not received data / heartbeat messages from the remote I/O device for 2000 milliseconds.

A remote I/O device consists of an iCAN device such as SmartStix modules that transmit global data and receive directed network data.

### Inputs

#### ID – (TYPE: INT)

This is the network ID of the remote I/O from which to receive data. This can be a constant from 1 to 253 or can be a 16-bit register.

#### DST[ ] – (TYPE: INT[])

This is the location to start placing received data from the remote I/O device. The number of registers used is defined by the "#N" parameter (see below).

#### #N – (TYPE: INT)

This is the number of words to receive from the remote I/O device. Typically for digital devices a 1 to 16 point module requires 1 word of data, a 17-32-point module requires 2 words.

### Outputs

#### Status – (TYPE: INT)

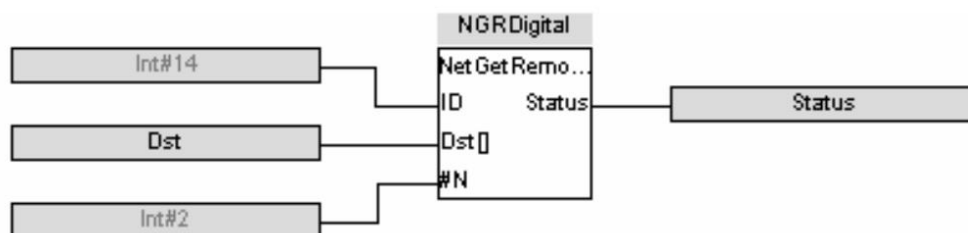
This 16-bit register is used internally. It should not be written by any other function block. Use the power flow from this function for the pass/fail status.

ST Language

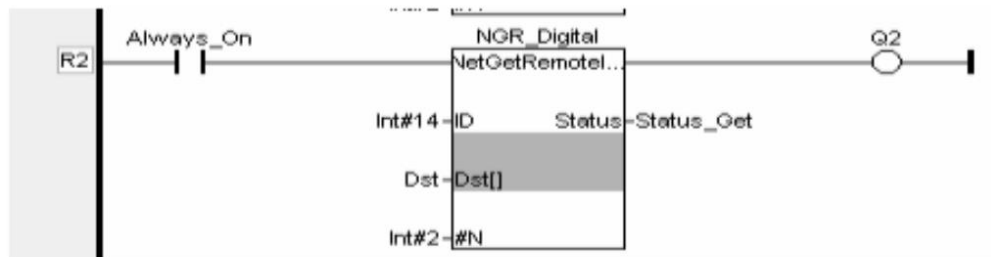
(\* NGRID is a declared instance of NetGetRemoteIO\_D function block \*)

```
NGRID(ID, DST[], #N);  
STATUS := NGRID.Status;
```

### FBD Language



### LD Language



IL Language

(\* NGRID is a declared instance of NetGetRemoteIO\_D function block \*)

Op1: CAL NGRID(ID, DST[], #N)

LD NGRID.Status

ST STATUS

**See also**

NetGetW NetGetHB NetGetRemoteIO\_A



## Net Get Words

Operator - This element allows global data from any device on the network to be copied into any set of registers. If the device defined by the source ID has not transmitted data this function block will not pass power flow and will send a request for the data to be sent. Once the requested data has been received, power flow from this function block will turn on.

This function works with either iCAN or DeviceNet networks.

### Inputs

#### ID – (TYPE: INT)

This register or constant defines the source node for the global data. If the ID is not valid, the function will do nothing and will not pass power.

#### @SRC – (TYPE: INT, BOOL)

This defines the starting point for the requested global data. This can be a %AQG or %QG registers. Note that %QG registers must be on a word boundary (1, 17, 33...).

This is a network register, a register assigned and produced by the transmitting ID.

#### DST[ ] – (TYPE: INT[], BOOL[])

This defines the starting register for the destination of the data. This is a register in the local controller.

#### #N – (TYPE: INT)

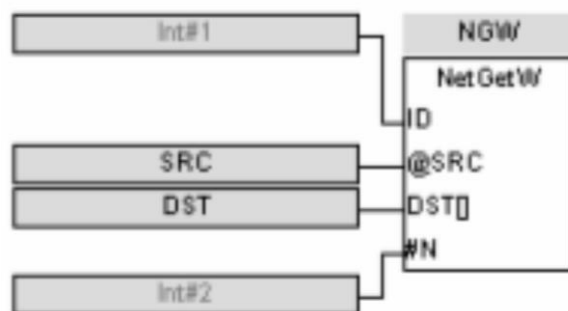
This defines the number of words to get from the source ID. The valid range is 1 to 32.

## ST Language

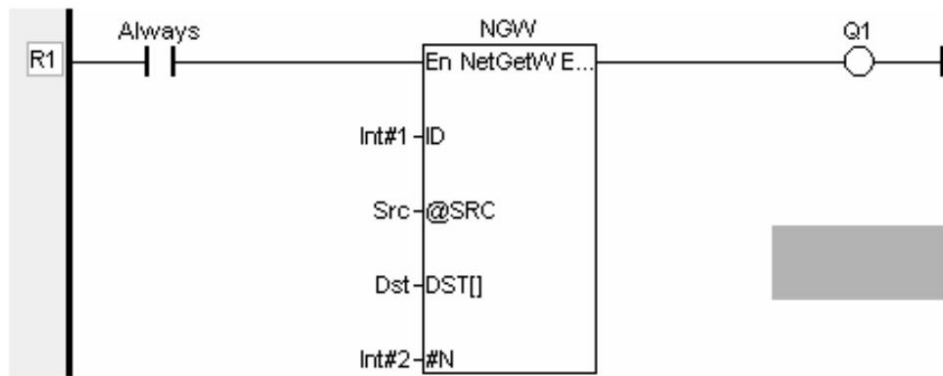
(\* NGW is a declared instance of NetGetW function block \*)

NGW (ID, @SRC, DST[], #N);

## FBD Language



## LD Language



### ***IL Language***

(\* NGW is a declared instance of NetGetW function block \*)

Op1: CAL NGW (ID, @SRC, DST[], #N)

### ***See also***

NetGetHB NetGetRemoteIO\_A NetGetRemoteIO\_D

## Net Put Heartbeat

This function allows a device to transmit a heartbeat iCAN message at a given rate to indicate to other devices it is on-line and operating normally. This function does generate network traffic. The message generated normally does not affect bandwidth, but if many devices send heartbeat messages frequently it may cause reduction in bandwidth.

This function will not pass power flow if the ID is not in the legal range.

This function works only with iCAN networks.

### Inputs

EN – (TYPE: BOOL)

The "EN" input is a condition. If EN is TRUE State then ENO is same.

ID – (TYPE: INT)

This register or constant is usually the primary network ID of the device (%SR29), but can be in the range defined by the primary network ID and the total number of IDs assigned to this device.

#PT – (TYPE: INT)

This is how often in milliseconds to send the heartbeat message. This has a range of 1 to 6553.

### Outputs

Status – (TYPE: INT)

This register is currently used for internal record keeping. Do not allow other function to write to this register.

ENO – (TYPE: BOOL)

ENO is equal to EN

### ST Language

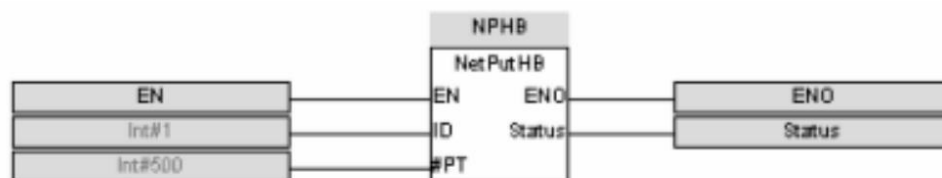
(\* NPHB is a declared instance of NetPutHB function block \*)

```
NPHB(EN, ID, #PT);
```

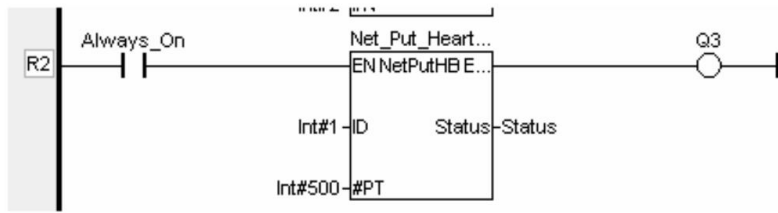
```
ENO := NPHB.ENO;
```

```
STATUS := NPHB.Status;
```

### FBD Language



### LD Language



### ***IL Language***

(\* NPHB is a declared instance of NetPutHB function block \*)

```
Op1: CAL NPHB(EN, ID, #PT)
      LD NPHB.ENO
      ST ENO
      LD NPHB.STATUS
      ST STATUS
```

### ***See also***

NetPutW   NetPutWex   NetPutRemoteIO\_A   NetPutRemoteIO\_D

### **Net Put Remote I/O Function Block (For Analog)**

**Operator** - This function handles sending data to a remote I/O function block and gets the sent data from a set of registers specified by the user. This function passes power flow if the remote I/O device is function normally. This function does not pass power flow if the remote I/O device has not sent a heartbeat in 2000 milliseconds. This function sends heartbeat messages to the output device every 1000 milliseconds. The default remote I/O operation is to expect heartbeat messages at least every 2000 milliseconds otherwise the outputs are turned off (or their configured default state). Data is normally transmitted on change of state, if the remote I/O device loses power, the I/O state is also sent when it resumes operation. A remote I/O device consists of a iCAN device such as SmartStix modules that transmit global data and receive directed network data.

#### ***Inputs***

##### **ID – (TYPE: INT)**

This is the network ID of the remote I/O to direct the sent data. This can be a constant from 1 to 253 or can be a 16-bit register.

##### **SRC[ ] – (TYPE: INT[])**

This is the starting location to get data to send to the remote I/O device. When this data changes state, it is sent to the remote I/O device. The number of registers used is defined by the "#N" parameter (see below).

##### **#N – (TYPE: INT)**

This is the number of words to send to the remote I/O device. Typically for digital devices a 1 to 16 point module requires 1 word of data, a 17-32-point module requires 2 words.

#### ***Outputs***

##### **Status – (TYPE: INT)**

This 16-bit register is used internally. It should be not be written by any other function block.

Bit 1-12 - reserved or internal use only

Bit 13 - Remote I/O OK and in sync with supplied data

Bit 14 - the Remote I/O detected a heartbeat error

Bit 15 - the Remote I/O has just powered up

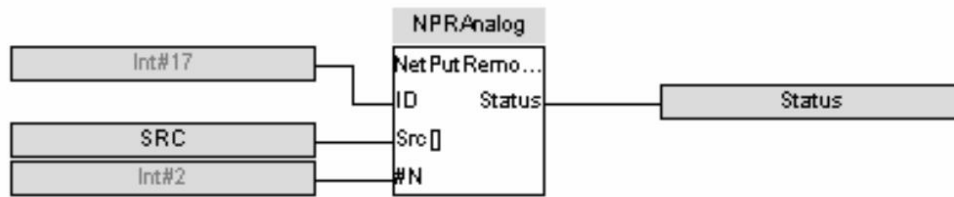
Bit 16 - the function is forcing a send (unit just power cycled or first scan)

ST Language

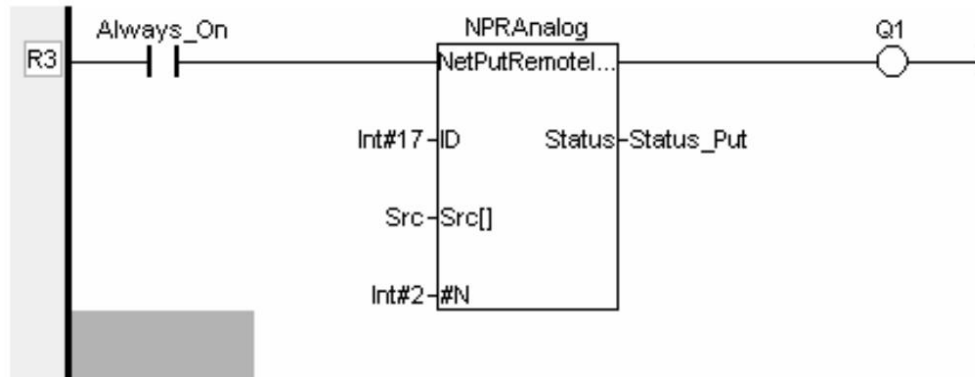
(\* NPRIA is a declared instance of NetPutRemoteIO\_A function block \*)

```
NPRIA(ID, SRC[], #N);  
STATUS := NPRIA.Status;
```

#### ***FBD Language***



### ***LD Language***



### ***IL Language***

(\* NPRIA is a declared instance of NetPutRemoteIO\_A function block \*)

```

Op1: CAL NPRIA(ID, SRC[], #N)
LD NPRIA.STATUS
ST STATUS

```

### ***See also***

NetPutW NetPutWex NetPutHB NetPutRemoteIO\_D

### **Net Put Remote I/O Function Block (For Digital)**

**Operator** - This function handles sending data to a remote I/O function block and gets the sent data from a set of registers specified by the user. This function passes power flow if the remote I/O device is function normally. This function does not pass power flow if the remote I/O device has not sent a heartbeat in 2000 milliseconds. This function sends heartbeat messages to the output device every 1000 milliseconds. The default remote I/O operation is to expect heartbeat messages at least every 2000 milliseconds otherwise the outputs are turned off (or their configured default state). Data is normally transmitted on change of state, if the remote I/O device loses power, the I/O state is also sent when it resumes operation. A remote I/O device consists of a iCAN device such as SmartStix modules that transmit global data and receive directed network data.

#### **Inputs**

##### **ID – (TYPE: INT)**

This is the network ID of the remote I/O to direct the sent data. This can be a constant from 1 to 253 or can be a 16-bit register.

##### **SRC[ ] – (TYPE: INT[], BOOL[])**

This is the starting location to get data to send to the remote I/O device. When this data changes state, it is sent to the remote I/O device. The number of registers used is defined by the "# N" parameter (see below).

##### **# N – (TYPE: INT)**

This is the number of words to send to the remote I/O device. Typically for digital devices a 1 to 16 point module requires 1 word of data, a 17-32-point module requires 2 words.

#### **Outputs**

##### **Status – (TYPE: INT)**

This 16-bit register is used internally. It should be not be written by any other function block.

Bit 1-12 - reserved or internal use only

Bit 13 - Remote I/O OK and in sync with supplied data

Bit 14 - the Remote I/O detected a heartbeat error

Bit 15 - the Remote I/O has just powered up

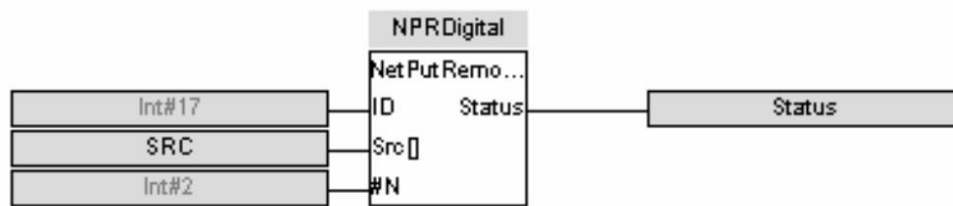
Bit 16 - the function is forcing a send (unit just power cycled or first scan)

ST Language

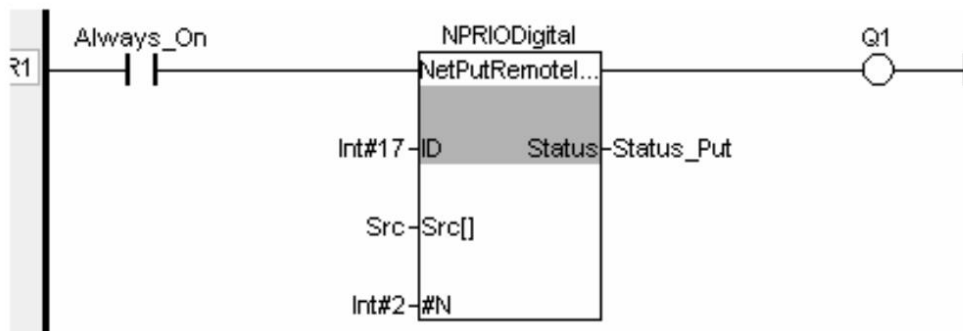
(\* NPRID is a declared instance of NetPutRemoteIO\_D function block \*)

```
NPRID(ID, SRC[], #N);  
STATUS := NPRID.Status;
```

#### **FBD Language**



### LD Language



### IL Language

(\* NPRID is a declared instance of NetPutRemoteIO\_D function block \*)

```
Op1: CAL NPRID(ID, SRC[], #N)
      LD NPRID.STATUS
      ST STATUS
```

### See also

NetPutW NetPutWex NetPutHB NetPutRemoteIO\_A



## Net Put Words

Operator - This element allows sending global data from any set of registers to any device on the network using multiple network IDs. If source data is not transmitted the function block will not pass power. Once the data is transmitted, function block will pass power.

This function works with iCAN network.

### Inputs

#### ID – (Type: INT)

ID - This is a register or constant for the ID to use when transmitting data on the network. It must be in the range defined by the primary network ID and the total nodes allocated for this target.

#### SRC[ ] – (Type: INT[], BOOL[])

This is the starting register for the source data to send on the network. This is a register local to the controller.

#### @DST - (Type: INT, BOOL)

This is the starting register for the destination of the data. Note that %QG registers must be on a word boundary (1, 17, 33...). This is a network register assigned to the network ID.

#### #N - (Type: INT)

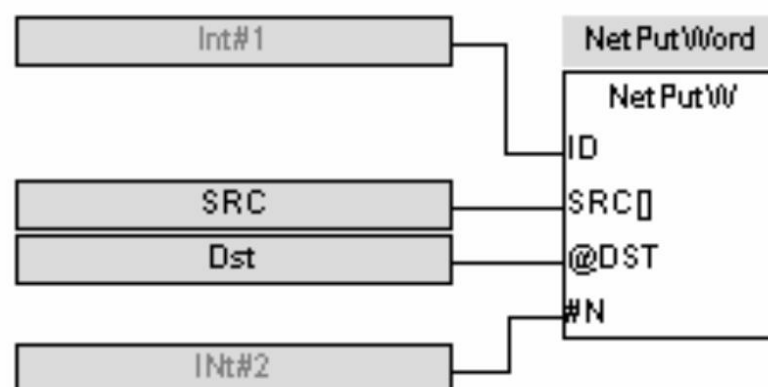
This is the number of words to send on the network.

### ST Language

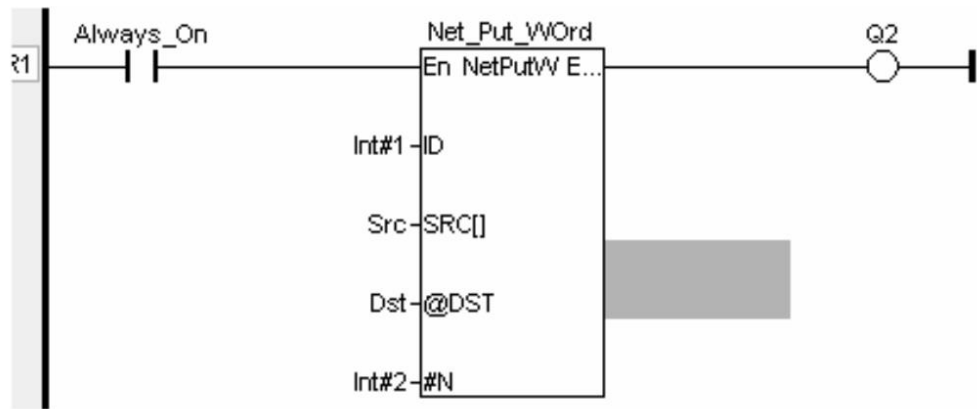
(\* NPW is a declared instance of NetPutW function block \*)

```
NPW(ID, SRC[], @DST, #N);
```

### FBD Language



### LD Language



IL Language

(\* NPW is a declared instance of NetPutW function block \*)

Op1: CAL NPW(ID, SRC[], @DST, #N)

### **See also**

NetPutWex NetPutHB NetPutRemoteIO\_A NetPutRemoteIO\_D

## **Net Put Wordex**

*Operator* – This element allows sending global data using multiple network IDs. When the "Send on Change of State" option is not checked, this function will copy the data from the source registers and attempt to transmit the data every scan that this function receives power. When the "Send on Change of State" option is checked and the SEND trigger is set LOW while the function receives power, this function will copy data from the source registers and attempt to transmit the data only if it changes. When the "Send on Change of State" option is checked and the SEND trigger is set HIGH while the function receives power, data will be sent every scan that the function receives power.

This function only works with iCAN networks.

The function passes power if the ID is legal and in the range defined by the network ID and the total number of ID assigned to that node.

### **Inputs**

#### **ID – (Type: INT)**

ID - This is a register or constant for the ID to use when transmitting data on the network. It must be in the range defined by the primary network ID and the total nodes allocated for this target.

#### **SRC[ ] – (Type: INT[], BOOL[])**

This is the starting register for the source data to send on the network. This is a register local to the controller.

#### **@DST - (Type: INT, BOOL)**

This is the starting register for the destination of the data. Note that % QG registers must be on a word boundary (1, 17, 33...). This is a network register assigned to the network ID.

#### **# N - (Type: INT)**

This is the number of words to send on the network.

#### **@SEND – (TYPE: BOOL)**

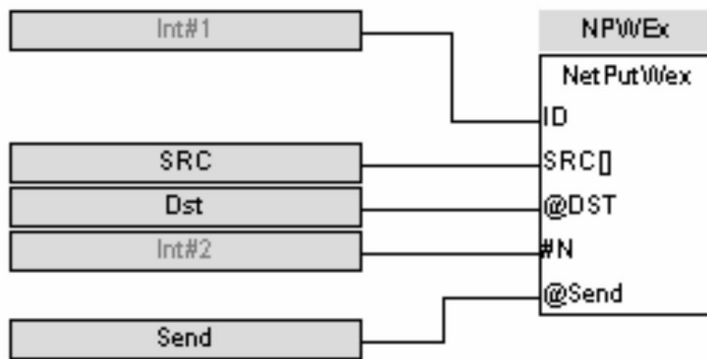
If the option to "Send on Change of State" is checked within the Net Put configuration, a Send trigger must be configured. This registers, when high, forces a data transmission and ignores the Change of State.

### **ST Language**

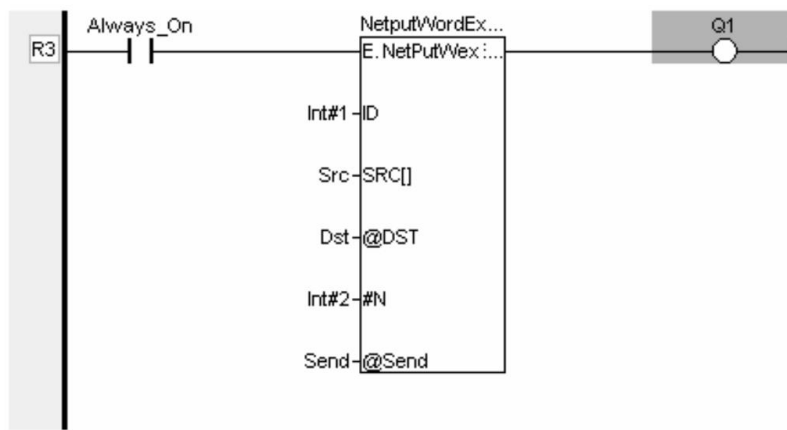
(\* NPWEX is a declared instance of NetPutWex function block \*)

NPWEX(ID, SRC[], @ DST, # N, @SEND);

### **FBD Language**



### LD Language



### IL Language

(\* NPWEX is a declared instance of NetPutWex function block \*)

Op1 : CAL NPWEX(ID, SRC[], @ DST, # N, @SEND)

### See also

NetPutW NetPutHB NetPutRemoteIO\_A NetPutRemoteIO\_D

## Floating PID Operations

### *Floating PID Operations*

Below are the standard blocks for Floating PID operations:

PID_IND_Real	PID independent
PID_IND_Auto_Real	PID independent with Auto Tuning feature
PID_ISA_Real	PID ISA
PID_ISA_Auto_Real	PID ISA with Auto Tuning feature
SETPID_Real	Sets PID registers
SETPIDMAN_R	

## PID\_IND\_R

*Operator* - Performs the proportional integral derivative (PID) IND algorithm for real SP, PV & to output a real CV value.

### Inputs

**CB[ ]:** Input indicating the location of a control block used to maintain the PID state for this PID loop. (TYPE : INT[])

**SP:** Process Setpoint (TYPE : REAL)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may be a decimal constant.

**PV:** Process Variable (TYPE : REAL)

Enter a register address or select a named register. This is the location of the Process Variable value coming in from the process.

**MAN EN:** Manual / Auto Boolean Switch (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register.

**UP:** Manual Mode up adjustment input (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register.

**DOWN:** Manual Mode down adjustment (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register.

**MUL:** This selects the precision of the real inputs used in the PID loops. For example if accuracy to 0.01 is required in the loop, select 0.01. (TYPE : REAL)

### Outputs

**CV:** The control variable result (TYPE : REAL)

Enter a register address, or select a named register. This is the location of the Control Variable value going out to the process.

### Remarks

**Independent PID Real PID\_INT\_R:**

**CVout = (Kp \* Error) + (Ki \* Error \* dt) + (Kd \* Derivative) + CVBias**

Where:

dt = Internal elapsed time clock - previous elapsed time clock

Derivative = (Error - previous Error)/dt

--or--

Derivative = (pv - previous PV)/dt

[User selectable during configuration].

Ti = Integral time

Td = Derivative time

Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will presumably be of type %R. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>
11	Manual Command	CV Counts	Tracks CV in Auto mode; sets CV in Manual Mode.	In the Automatic mode this register tracks the CV value. In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.

12	Internal SP	Used by <i>i</i> <sup>3</sup>	N/A	Tracks SP in multiplied by the input MUL
13	Internal PV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks PV in multiplied by the input MUL
14	Internal CV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks CV out multiplied by the input MUL
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

Registers at offset 0 through 9 must be configured before the PID element is used.

**Examples:** PID\_IND\_R1 is a declared instance of PID\_IND\_R function block.

#### ST Language

```
PID_IND_R1(CB[], SP, PV, MANEN, UP, DOWN, MUL);
CV := PID_IND_R1.CV;
```

#### FBD Language

##### LD Language

En is the Enable input & Eno is the enable output. En & Eno will be the same state.

The other functionality is similar to that of a FBD.

#### IL Language

(\*PID\_IND\_R1 is a declared instance of PID\_IND\_R function block \*)

```
Op1: CAL PID_IND_R1(CB[], SP, PV, MANEN, UP, DOWN, MUL)
LD PID_IND_R1.CV
ST CV
```

Caution: Overlapping references will result in erratic operation of the PID algorithm.

See also

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)



## **PID\_IND\_AUTO\_R**

*Operator* - Performs the proportional integral derivative (PID) IND algorithm with auto tuning function for real SP, PV & to output a real CV value.

### **Inputs**

CB[ ]: Input indicating the location of a control block used to maintain the PID state for this PID loop. (TYPE : INT[])

SP: Process Setpoint (TYPE : REAL)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may be a decimal constant.

PV: Process Variable (TYPE : REAL)

Enter a register address or select a named register. This is the of the Process Variable value coming in from the process.

MAN EN: Manual / Auto Boolean Switch (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register.

UP: Manual Mode up adjustment input (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register.

DOWN: Manual Mode down adjustment (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register.

TUNE: Input which controls when the function should start the auto tune process. (TYPE : BOOL)

A boolean TUNE input starts the autotuning cycle. This input needs to be held high during the autotuning cycle. If it is negated during the AUTOTUNE cycle, the controller stops autotune and reverts to the previous settings.

#FILTER : (TYPE : DINT)

Allowable inputs:

FILTER\_0\_04 – Filters at 0.04%

FILTER\_0\_08 – Filters at 0.08%

FILTER\_0\_16 – Filters at 0.16%

FILTER\_0\_31 – Filters at 0.31%

FILTER\_0\_63 – Filters at 0.63%

FILTER\_1\_25 – Filters at 1.25%

FILTER\_2\_50 – Filters at 2.5%

FILTER\_5\_00 – Filters at 5%

This input defines how far above and below the setpoint the process must go when performing the auto tune experiment. Hysteresis is applied to the setpoint using the

selected filter constant – if the process is subject to noise it is recommended that the process autotune is setup with a higher percentage. Higher noise rejection filters will also cause the autotuning algorithm to select slightly slower more stable coefficients. Where the process is noisy it is recommended that PI rather than PID control is selected.

#RESP : ( TYPE : DINT)  
Allowable Inputs  
PID\_FAST  
PID\_MEDIUM  
PID\_SLOW  
PID\_VERYSLOW

This defines the relative speed of the PID loop once it is tuned.

#TYPE : (TYPE : DINT)  
Allowable Values:  
TYPE\_PID  
TYPE\_PI  
TYPE\_P

This option allows the auto tune procedure to calculate terms for PID, PI or P terms.

#TUNE2/3 : (TYPE : BOOL)

This allows the auto tuning experiment to change the output based on 2/3 the set point. Use this option when it is not desired for the process to travel above the setpoint during the auto tuning experiment.

MUL: This selects the precision of the real inputs used in the PID loops. For example if accuracy to 0.01 is required in the loop, select 0.01. (TYPE : REAL).

### Outputs

CV: The control variable result (TYPE : REAL)

Enter a register address, or select a named register. This is the location of the Control Variable value going out to the process.

DONE: The control variable result (TYPE : BOOL)

This defines an output bit that is set by the function when the auto tune is complete.

### Remarks

Independent PID\_Auto\_R:

$CV_{out} = (K_p * Error) + (K_i * Error * dt) + (K_d * Derivative) + CV_{Bias}$

Where:

$dt = \text{Internal elapsed time clock} - \text{previous elapsed time clock}$

$Derivative = (Error - \text{previous Error})/dt$

--or--

$Derivative = (pv - \text{previous PV})/dt$

[User selectable during configuration].

Ti = Integral time

Td = Derivative time

### Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will presumably be of type %R. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>
11	Manual Command	CV Counts	Tracks CV in Auto mode; sets	In the Automatic mode this register tracks the CV value.

			CV in Manual Mode.	In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.
12	Internal SP	Used by $i^3$	N/A	Tracks SP in multiplied by the input MUL
13	Internal PV	Used by $i^3$	N/A	Tracks PV in multiplied by the input MUL
14	Internal CV	Used by $i^3$	N/A	Tracks CV out multiplied by the input MUL
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

Registers at offset 0 through 9 must be configured before the PID element is used.

**Examples:** PID\_IND\_Auto\_R1 is a declared instance of PID\_IND\_Auto\_R function block

#### ST Language

```
PID_IND_Auto_R1(CB[], SP, PV, MANEN, UP, DOWN, TUNE, #FILTER, #RESP,
#TYPE, #TUNE2/3, MUL);
CV := PID_IND_Auto_R1.CV ;
Done := PID_IND_Auto_R1.Done;
```

#### FBD Language

#### LD Language

En is the Enable input & Eno is the enable output. En & Eno will be the same state.

The other functionality is similar to that of a FBD.

#### IL Language

```
Op1: CAL PID_IND_Auto_R1(CB[], SP, PV, MANEN, UP, DOWN, TUNE,
#FILTER, #RESP, #TYPE, #TUNE2/3, MUL)
LD PID_IND_Auto_R1.CV
ST CV
LD PID_IND_Auto_R1.Done
ST Done
```

**Caution:** Overlapping references will result in erratic operation of the PID algorithm.

**See also**

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## **PID\_ISA\_R**

*Operator* - Performs the proportional integral derivative (PID) IND algorithm for real SP, PV & to output a real CV value.

### **Inputs**

CB[ ]: Input the values of the register usage mentioned below. (TYPE : INT[])

SP: Process Setpoint (TYPE : REAL)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may NOT be a decimal constant.

PV: Process Variable (TYPE : REAL)

Enter a register address or select a named register. This is the location (typically %AI) of the Process Variable value coming in from the process. This value may NOT be a decimal constant.

MAN EN: Manual / Auto Boolean Switch (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register, typically %T.

UP: Manual Mode up adjustment input (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register, typically %T.

DOWN: Manual Mode down adjustment (TYPE : BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register, typically %T.

**MUL:** This selects the precision of the real inputs used in the PID loops. For example if accuracy to 0.01 is required in the loop, select 0.01. (TYPE : REAL)

### **Outputs**

CV: The control variable result (TYPE : REAL)

Enter a register address, or select a named register. This is the location (typically %AQ) of the Control Variable value going out to the process. This value may NOT be a decimal constant.

### **Remarks**

#### **PID\_ISA\_R :**

**CVout = Kp \* (Error + (Error \* dt / Ti) + (Td \* Derivative)) + CVBias**

Where:

dt = Internal elapsed time clock - previous elapsed time clock

Derivative = (Error - previous Error)/dt

--or--

Derivative = (pv - previous PV)/dt

[User selectable during configuration].

Ti = Integral time

Td = Derivative time

### **Register Usage**

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>

11	Manual Command	CV Counts	Tracks CV in Auto mode; sets CV in Manual Mode.	In the Automatic mode this register tracks the CV value. In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.
12	Internal SP	Used by <i>i<sup>3</sup></i>	N/A	Tracks SP in
13	Internal PV	Used by <i>i<sup>3</sup></i>	N/A	Tracks PV in
14	Internal CV	Used by <i>i<sup>3</sup></i>	N/A	Tracks CV out
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

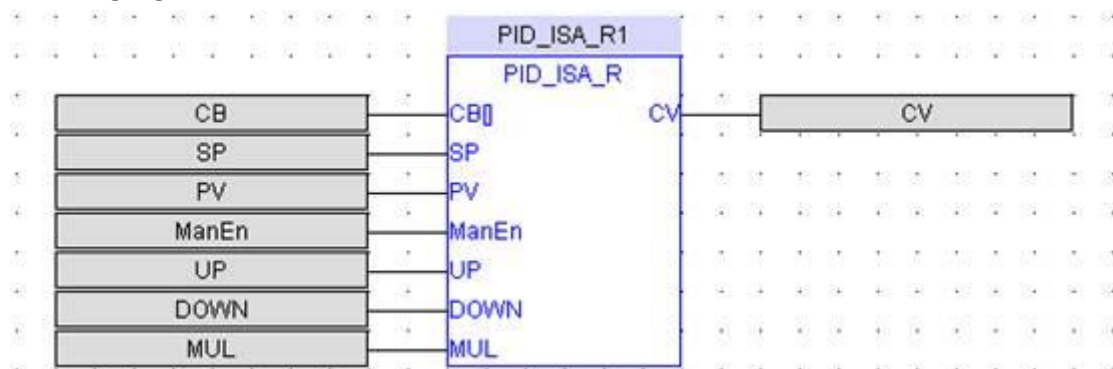
Registers at offset 0 through 9 must be configured before the PID element is used.

**Examples:** PID\_ISA\_R1 is a declared instance of PID\_ISA\_R function block

#### ST Language

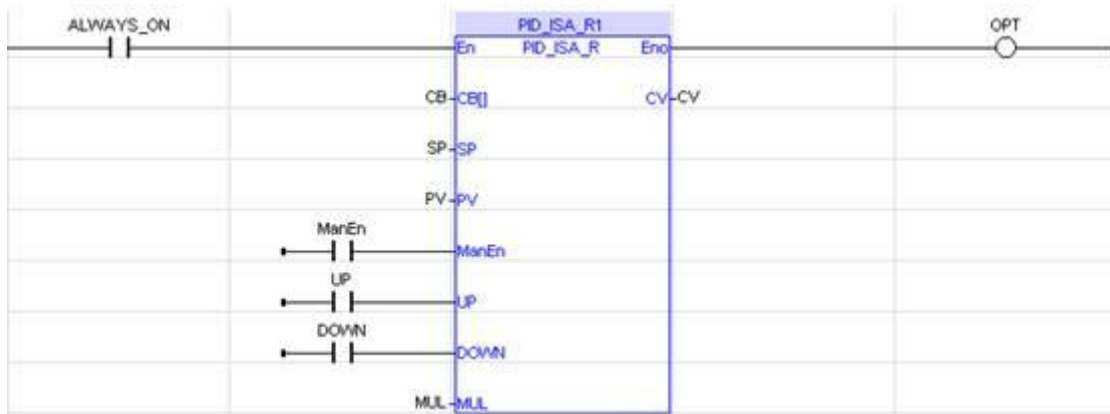
```
PID_ISA_R1(CB[], SP, PV, MANEN, UP, DOWN);
CV := PID_ISA_R1.CV;
```

#### FBD Language



#### LD Language





En is the Enable input & Eno is the enable output. En & Eno will be the same state.

The other functionality is similar to that of a FBD.

### IL Language

```
Op1: CAL PID_ISA_R1(CB[], SP, PV, MANEN, UP, DOWN)
LD PID_ISA_R1.CV
ST CV
```

**Caution:** Overlapping references will result in erratic operation of the PID algorithm.

### See also

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## PID\_ISA\_AUTO\_R

*Operator* - Performs the proportional integral derivative (PID) IND algorithm with auto tuning function. For real SP, PV & to output a real CV value.

Inputs

**CB[ ]:** Input the values of the register usage mentioned below. (TYPE: INT[])

**SP:** Process Setpoint (TYPE: REAL)

Enter a register address, or select a named register. This is the location of the User-defined Process Setpoint value. This value may NOT be a decimal constant.

**PV:** Process Variable (TYPE: REAL)

Enter a register address or select a named register. This is the location (typically **%AI**) of the Process Variable value coming in from the process. This value may NOT be a decimal constant.

**MAN EN:** Manual / Auto Boolean Switch (TYPE: BOOL)

Enter a register address or select a named register that is the User-controlled Manual Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**UP:** Manual Mode up adjustment input (TYPE: BOOL)

Enter a register address or select a named register that is the User-controlled UP Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**DOWN:** Manual Mode down adjustment (TYPE: BOOL)

Enter a register address or select a named register that is the User-controlled DOWN Input bit. This register is a Boolean (1-bit) register, typically **%T**.

**TUNE:** Input which controls when the function should start the auto tune process. (TYPE: BOOL)

A boolean TUNE input starts the autotuning cycle. This input needs to be held high during the autotuning cycle. If it is negated during the AUTOTUNE cycle, the controller stops autotune and reverts to the previous settings.

**#FILTER:** (TYPE: DINT)

Allowable inputs:

FILTER\_0\_04 – Filters at 0.04%

FILTER\_0\_08 – Filters at 0.08%

FILTER\_0\_16 – Filters at 0.16%

FILTER\_0\_31 – Filters at 0.31%

FILTER\_0\_63 – Filters at 0.63%

FILTER\_1\_25 – Filters at 1.25%

FILTER\_2\_50 – Filters at 2.5%

FILTER\_5\_00 – Filters at 5%

This input defines how far above and below the setpoint the process must go when performing the auto tune experiment. Hysteresis is applied to the setpoint using the selected filter constant – if the process is subject to noise it is recommended that the process autotune is setup with a higher percentage. Higher noise rejection filters will also cause the autotuning algorithm to select slightly slower more stable coefficients.

Where the process is noisy it is recommended that PI rather than PID control is selected.

**#RESP:** (TYPE: DINT)

Allowable Inputs

PID\_FAST

PID\_MEDIUM

PID\_SLOW

PID\_VERYSLOW

This defines the relative speed of the PID loop once it is tuned.

**#TYPE:** (TYPE: DINT)

Allowable Values:

TYPE\_PID

TYPE\_PI

TYPE\_P

This option allows the auto tune procedure to calculate terms for PID, PI or P terms.

**#TUNE2/3:** (TYPE: BOOL)

This allows the auto tuning experiment to change the output based on 2/3 the set point. Use this option when it is not desired for the process to travel above the setpoint during the auto tuning experiment.

**MUL:** This selects the precision of the real inputs used in the PID loops. For example if accuracy to 0.01 is required in the loop, select 0.01. (TYPE: REAL)

### **Outputs**

**CV:** The control variable result (TYPE: REAL)

Enter a register address, or select a named register. This is the location (typically %AQ) of the Control Variable value going out to the process. This value may NOT be a decimal constant.

**DONE:** The control variable result (TYPE: BOOL)

This defines an output bit that is set by the function when the auto tune is complete.

### **Remarks**

**PID\_ISA\_Auto\_R :**

**CVout =  $K_p * (\text{Error} + (\text{Error} * dt / T_i) + (T_d * \text{Derivative})) + \text{CVBias}$**

Where:

dt = Internal elapsed time clock - previous elapsed time clock

Derivative = (Error - previous Error)/dt

--or--

Derivative = (pv - previous PV)/dt

[User selectable during configuration].

Ti = Integral time

Td = Derivative time

Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the Reference Array.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.
10	Config Word	N/A	N/A	<b>Internal Use - Do not modify this value.</b>

11	Manual Command	CV Counts	Tracks CV in Auto mode; sets CV in Manual Mode.	In the Automatic mode this register tracks the CV value. In the Manual Mode, this register contains the value that is output to the CV within the clamp and slew limits.
12	Internal SP	Used by <i>i</i> <sup>3</sup>	N/A	Tracks SP in
13	Internal PV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks PV in
14	Internal CV	Used by <i>i</i> <sup>3</sup>	N/A	Tracks CV out
15	Cycle Time	Seconds	N/A	Cycle Time for PWM in Seconds

Each PID element must use a distinctly separate Reference Array, even if the values are identical to an exiting PID element.

There can be no overlapping of PID elements.

Registers at offset 0 through 9 must be configured before the PID element is used.

**Examples:** PID\_ISA\_Auto\_R1 is a declared instance of PID\_ISA\_Auto\_R function block

ST Language

```
PID_ISA_Auto_R1(CB[], SP, PV, MANEN, UP, DOWN, TUNE, #FILTER, #RESP,
#TYPE, #TUNE2/3);
CV := PID_ISA_Auto_R1.CV;
DONE := PID_ISA_Auto_R1.Done;
```

FBD Language

### ***LD Language***

En is the Enable input & Eno is the enable output. En & Eno will be the same state.

The other functionality is similar to that of a FBD.

### ***IL Language***

```
Op1: CAL PID_ISA_Auto_R1(CB[], SP, PV, MANEN, UP, DOWN, TUNE,
#FILTER, #RESP, #TYPE, #TUNE2/3)
LD PID_ISA_Auto_R1.CV
ST CV
LD PID_ISA_Auto_R1.Done
ST DONE
```

**Caution:** Overlapping references will result in erratic operation of the PID algorithm.

***See also***

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## SETPID\_R

*Operator* - Performs the setting for PID Registers as Real data type for KP, KD, KI & to CV Bias, Upclamp and DnClamp value.

### Inputs

**CB[ ]:** Input the values of the register usage mentioned below (TYPE : INT[])

**PERIOD:** Sample Period (TYPE : INT)

**DEADBAND + :** Dead Band + (TYPE : INT)

Defines the Upper Dead Band limits in terms of PV counts.

Set to 0 (zero) if no dead band is required.

**DEADBAND - :** Dead Band - (TYPE : INT)

Defines the Upper Dead Band limits in terms of PV counts.

Set to 0 (zero) if no dead band is required.

Both Deadbands be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.

**Kp:** Proportional Gain (TYPE : REAL)

Sets the Proportional Gain (Kp) factor in terms of percent.

100 sets unity gain (gain of 1).

**Kd:** Derivative Gain (TYPE : REAL)

Entered as a time with a resolution of 10 mS.

In the PID equation this has the effect:  $K_d * \text{delta Error} / dt$ .

**Ki:** Integral Rate (TYPE : REAL)

Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect:  $K_i * \text{Error} * dt$ .

**CVBias:** CV Bias (TYPE : REAL)

Number of CV counts added to the output before the rate and amplitude clamps.

**CVUpClamp:** CV Upper Clamp (TYPE : REAL)

Number of CV Counts that represent the highest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.

**CVDnClamp:** CV Down Clamp (TYPE : REAL)

Number of CV Counts that represent the lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.

**MinSlew:** Minimum Slew Time (TYPE : INT)

Determines how fast the CV value can change.

**ErrAction:** Error Action (TYPE : BOOL)

**DAction:** (TYPE : BOOL)

**OutPol:** (TYPE : BOOL)

**Dsense:** (TYPE : BOOL)

**MUL:** This selects the precision of the real inputs used in the PID loops. For example if accuracy to 0.01 is required in the loop, select 0.01. (TYPE: REAL)

### Remarks

Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the reference Array.

Registers at offset 0 through 9 must be configured before the PID element is used.

This is configured using the SetPID block.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

**Examples:** SETPID\_R1 is a declared instance of SetPID\_R function block



### ***ST Language***

SETPID\_R1(CB[], Period, DeadBand +, Deadband-, Kp, Kd, Ki, CVBias, CVUpClamp, CVDnClamp, MinSlew, ErrAction, DAction, Outpol, Dsense, MUL);

### ***FBD Language***

#### ***LD Language***

En is the Enable input & Eno is the enable output. En & Eno will be the same state.

The other functionality is similar to that of a FBD.

#### ***IL Language***

Op1: CAL SETPID\_R1(CB[], Period, DeadBand+, Deadband-, Kp, Kd, Ki, CVBias, CVUpClamp, CVDnClamp, MinSlew, ErrAction, DAction, Outpol, Dsense, MUL)

**Caution:** Overlapping references will result in erratic operation of the PID algorithm.

#### ***See also***

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## SETPIDMAN\_R

*Operator:* Sets in Manual mode, the PID associated with the Control block with the Input to this block as Control Output value.

### Inputs

**CB[ ]:** Input the values of the register usage mentioned below (TYPE : INT[])

**IN:** Input value that will be fed to the Control Variable as Manual Output (TYPE: REAL)

**MUL:** This selects the precision of the real inputs used in the PID loops. For example if accuracy to 0.01 is required in the loop, select 0.01. (TYPE: REAL)

### Output

**OK:** Block execution indicator, it goes high if the block has been executed successfully.

### Remarks

#### Register Usage

Either PID element requires an array of fifteen (15) WORD (16-bit) registers. These will typically be of type %R. This is called the reference Array.

Registers at offset 0 through 9 must be configured before the PID element is used.

This is configured using the SETPID block.

Offset	Parameter	Units	Range	Description
0	Sample Period	10 mS	0 to 65535	The shortest time, in 10mS increments, allowed between PID solutions.
1	Dead Band +	PV Counts	0 to 32000	Defines the Upper and Lower Dead Band limits in terms of PV counts. Set both to 0 (zero) if no dead band is required. Both should be set to 0 (zero) until the PID is tuned. A Dead Band might then be necessary to prevent small changes in CV values due to slight variations in error.
2	Dead Band -	PV Counts	0 to 32000	
3	Proportional Gain (Kp)	Percent	0 to 327.67%	Sets the Proportional Gain (Kp) factor in terms of percent. 100 sets unity gain (gain of 1).
4	Derivative Gain (Kd)	10 mS	0 to 327.67 seconds	Entered as a time with a resolution of 10 mS. In the PID equation this has the effect: $K_d * \Delta \text{Error} / dt$ .

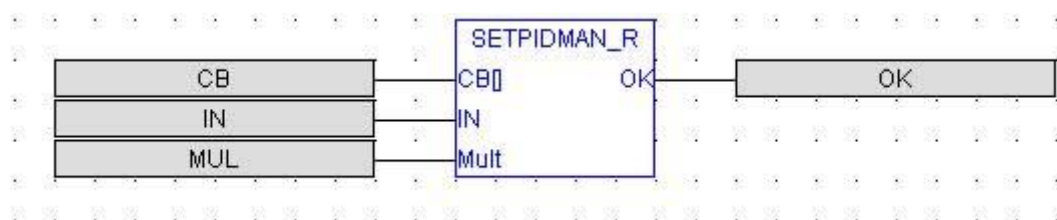
5	Integral Rate (Ki)	Repeats per 1000 second	0 to 32.767 repeats per second	Entered as a number of repeats per second -- effectively the integration rate. In the PID equation this has the effect: $K_i * \text{Error} * dt$ .
6	CV Bias	CV Counts	-32000 to +32000	Number of CV counts added to the output before the rate and amplitude clamps.
7	CV Upper Clamp	CV Counts	-32000 to +32000	Number of CV Counts that represent the highest and lowest value for CV. CV Upper Clamp must be more positive the CV Lower Clamp.
8	CV Lower Clamp	CV Counts	-32000 to +32000	
9	Minimum Slew Time	Seconds of full travel	0 to 32000 seconds to move 32000 CV counts	Determines how fast the CV value can change.

Each PID element must use a distinctly separate reference Array, even if the values are identical to an exiting PID element. There can be no overlapping of PID elements.

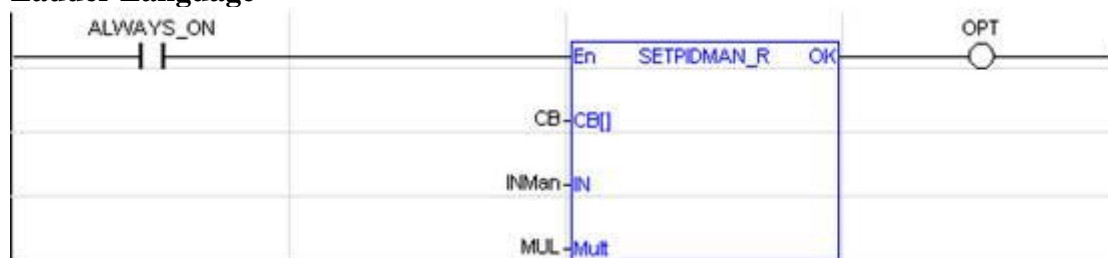
### ST Language

```
SETPIDMAN( CB, IN );
OK := SETPIDMAN. OK;
```

### FBD Language



### Ladder Language



### IL Language

```
OP1: CAL IN;
LD SETPIDMAN(CB)
ST Q
```

See also

[Overview\\_PID](#) | [Tuning\\_PID](#) | [Independent PID Loop](#) | [Independent PID with Auto Tune](#) | [ISA PID Loop](#) | [ISA PID with Auto Tune](#) | [Set PID Control Block](#) | [Set PID MAN](#)

## Timer Counter Operations

### *Timer Counter Operations*

Below are the standard functions for managing timers:

Off Timer - 100 ms Res.	Off delay timer with 100ms resolution
Off Timer - 100 ms Res. 32 Bit	Off delay timer with 100ms resolution 32 Bit
Off Timer - 10 ms Res.	Off delay timer with 10ms resolution
Off Timer - 10 ms Res. 32 Bit	Off delay timer with 10ms resolution 32 Bit
Off Timer - 1 ms Res.	Off delay timer with 1ms resolution
Off Timer - 1 ms Res. 32 Bit	Off delay timer with 1ms resolution 32 Bit
Off Timer - 1 Sec Res.	Off delay timer with 1 Sec resolution
Off Timer - 1 Sec Res. 32 Bit	Off delay timer with 1 Sec resolution 32 Bit
On Timer - 100 ms Res.	On delay timer with 100ms resolution
On Timer - 100 ms Res. 32 Bit	On delay timer with 100ms resolution 32 Bit
On Timer - 10 ms Res.	On delay timer with 10ms resolution
On Timer - 10 ms Res. 32 Bit	On delay timer with 10ms resolution 32 Bit
On Timer - 1 ms Res.	On delay timer with 1ms resolution
On Timer - 1 ms Res. 32 Bit	On delay timer with 1ms resolution 32 Bit
On Timer - 1 Sec Res.	On delay timer with 1 Sec resolution
On Timer - 1 Sec Res. 32 Bit	On delay timer with 1 Sec resolution 32 Bit
On Timer - 100 ms Res. Retentive	On delay timer with 100ms resolution Retentive
On Timer - 100 ms Res. 32 Bit Retentive	On delay timer with 100ms resolution 32 Bit Retentive
On Timer - 10 ms Res. Retentive	On delay timer with 10ms resolution Retentive
On Timer - 10 ms Res. 32 Bit Retentive	On delay timer with 10ms resolution 32 Bit Retentive
On Timer - 1 ms Res. Retentive	On delay timer with 1ms resolution Retentive
On Timer - 1 ms Res. 32 Bit Retentive	On delay timer with 1ms resolution 32 Bit Retentive
On Timer - 1 Sec Res. Retentive	On delay timer with 1 Sec resolution Retentive
On Timer - 1 Sec Res. 32 Bit Retentive	On delay timer with 1 Sec resolution 32 Bit Retentive

### **TOF100mS / TOF10mS/TOF1mS/TOF1Sec - (Res. 32 Bit)**

*Operator* - Performs OFF delay timer operations using these blocks with specified timebase.

#### **Resolution:**

16 BIT Resolution Timers: The count value limit in these timers is 65535.

32 BIT Resolution Timers: The 32 –bit resolution Timers are useful in applications where the count value needs is insufficient using the regular timers of 16 bit. The count value limit in these timers is 4294967295.

#### **Inputs**

IN : Input for resetting Count up time (CT) & enable the output Q. (TYPE : BOOL)

PT : Programmed time, Maximum count up for Count up time (CT). (TYPE : INT)

#### **Outputs**

Q : Output which stays TRUE till the count up is active & goes FALSE when CT=PT. (TYPE : BOOL)

CT : The counter for the specified timebase, to reach the Programmed time. (TYPE : INT)

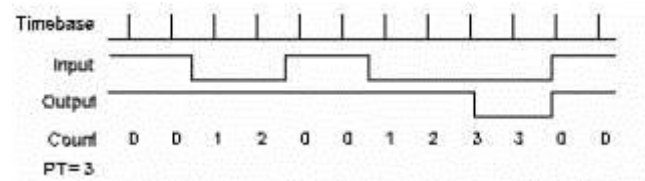
#### **Remarks**

The timer starts on a falling pulse of IN input. It stops when the Count up time (CT) is equal to the programmed time (PT). A rising pulse of IN input resets the timer to 0.

The output signal is set to TRUE when the IN input rises to TRUE, reset to FALSE when programmed time is elapsed for the specified timebase.

In LD language, the input rung is the IN command. The output rung is the Q output signal.

The timebase is user definable in 10mS or 100mS ticks. When input IN goes high the counting proceeds based on the timebase block used.



#### **ST Language**

(\* TOF1 is a declared instance of TOF100ms function block \*)

TOF1 (IN, PT);

Q := TOF1.Q;

CT := TOF1.CT;

**or**

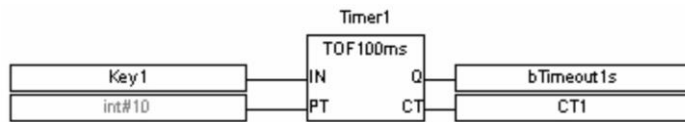
(\* TOF1 is a declared instance of TOF10ms function block \*)

TOF1 (IN, PT);

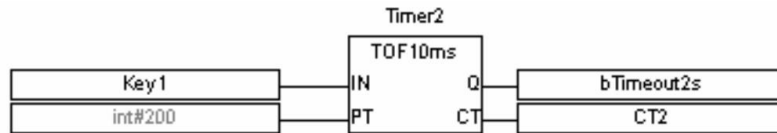
Q := TOF1.Q;

CT := TOF1.CT;

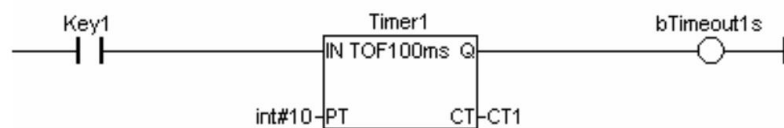
#### **FBD Language**



**or**



### **LD Language**



**or**

### **IL Language**

(\* TOF1 is a declared instance of TOF100ms function block \*)

Op1: CAL TOF1 (IN, PT)

LD TOF1.Q

ST Q

LD TOF1.CT

ST CT

**or**

(\* TOF1 is a declared instance of TOF10ms function block \*)

Op1: CAL TOF1 (IN, PT)

LD TOF1.Q

ST Q

LD TOF1.CT

ST CT

### **See also**

TON100ms TON10ms TONR100ms TONR10ms

### **TON100mS / TON10mS/TON1mS / TON1Sec – (Res. 32 Bit)**

Operator - Performs ON delay timer operations using these blocks with specified time base.

#### **Resolution:**

16 BIT Resolution Timers: The count value limit in these timers is 65535.

32 BIT Resolution Timers: The 32 –bit resolution Timers are useful in applications where the count value needs is insufficient using the regular timers of 16 bit. The count value limit in these timers is 4294967295.

#### **Inputs**

IN : Input for starting Count up time (CT).(TYPE : BOOL)

PT : Programmed time, Maximum count up for CT. (TYPE : INT)

#### **Outputs**

Q : Output goes TRUE when CT = PT & goes FALSE when input is low. (TYPE : BOOL)

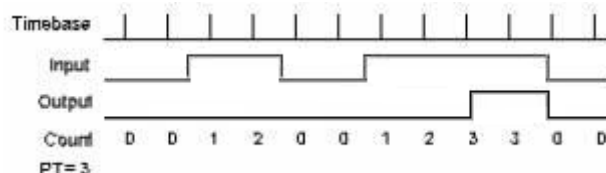
CT : The counter of the timebase specified, to reach the Programmed time. (TYPE : INT)

#### **Remarks**

The timer starts on a rising pulse of IN input. It stops when the Count up time (CT) is equal to programmed time (PT). A falling pulse of IN input resets the timer to 0. The output signal is set to TRUE when programmed time is elapsed, and reset to FALSE when the input command falls.

In LD language, the input rung is the IN command. The output rung is Q the output signal.

The timebase is user definable in 10mS or 100mS "ticks". When input IN goes high the counting proceeds based on the timebase block used.



#### **ST Language**

(\* TON1 is a declared instance of TON100ms function block \*)

TON1 (IN, PT);

Q := TON1.Q;

CT := TON1.CT;

**or**

(\* TON1 is a declared instance of TON10ms function block \*)

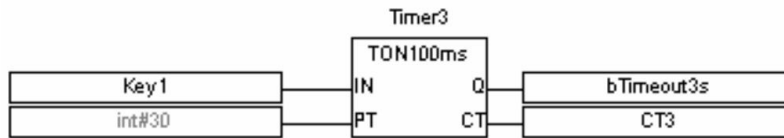
TON1 (IN, PT);

Q := TON1.Q;

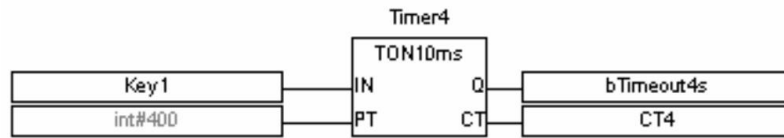
CT := TON1.CT;

#### **FBD Language**

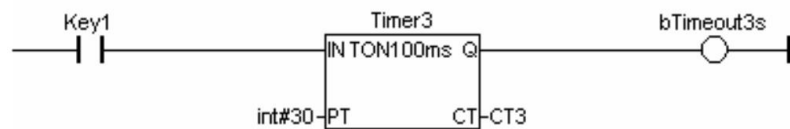




**or**



### **LD Language**



**or**

### **IL Language**

(\* TON1 is a declared instance of TON100ms function block \*)

Op1: CAL TON1 (IN, PT)

LD TON1.Q

ST Q

LD TON1.CT

ST CT

**OR**

(\* TON1 is a declared instance of TON10ms function block \*)

Op1: CAL TON1 (IN, PT)

LD TON1.Q

ST Q

LD TON1.CT

ST CT

### **See also**

TOF100ms TOF10ms TONR100ms TONR10ms

## **TONR100mS / TONR10mS/TONR1mS / TONR1Sec – (Res. 32 Bit)**

Operator - Performs retentive ON timer operations using these blocks.

### **Resolution:**

16 BIT Resolution Timers: The count value limit in these timers is 65535.

32 BIT Resolution Timers: The 32 –bit resolution Timers are useful in applications where the count value needs is insufficient using the regular timers of 16 bit. The count value limit in these timers is 4294967295.

### **Inputs**

IN : Timer command. (TYPE : BOOL)

RESET : For resetting the counter CT & output Q. (TYPE : BOOL)

PT : Preset time, Maximum count up for CT. (TYPE : INT)

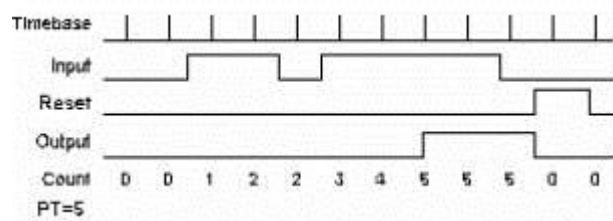
### **Outputs**

Q : Output goes TRUE when CT = PT & goes FALSE when reset is pressed (TYPE : BOOL)

CT : The counter of the timebase specified, to reach the Programmed time. Set to zero if reset is pressed. (TYPE : INT )

### **Remarks**

A Retentive On Delay Timer is a special case of the "standard" On Delay Timer. It differs from the standard timer in that the Retentive Timer does *not* reset when the input is brought inactive (off). The Retentive Timer requires that a reset signal be applied to the element in order for the timer to be reset.



### **ST Language**

(\* TONR1 is a declared instance of TONR100ms function block \*)

TONR1 (IN,RESET,PT);

Q := TONR1.Q;

CT := TONR1.CT;

**or**

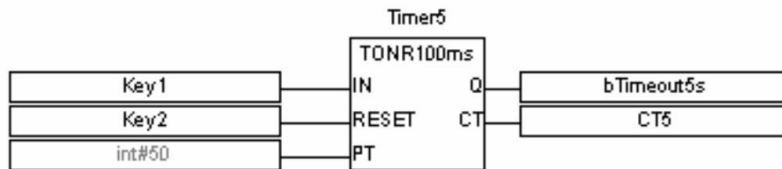
(\* TONR1 is a declared instance of TONR10ms function block \*)

TONR1 (IN,RESET,PT);

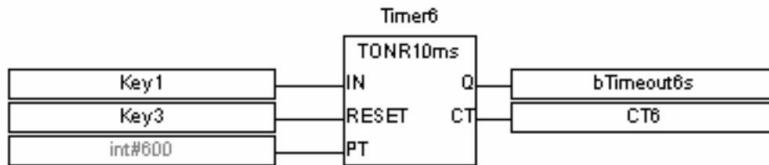
Q := TONR1.Q;

CT := TONR1.CT;

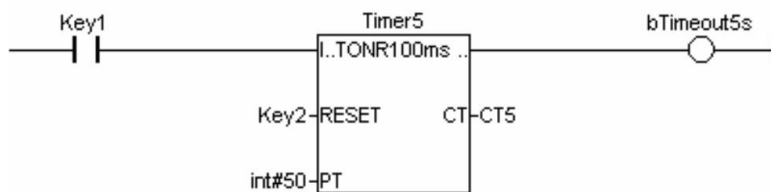
### **FBD Language**



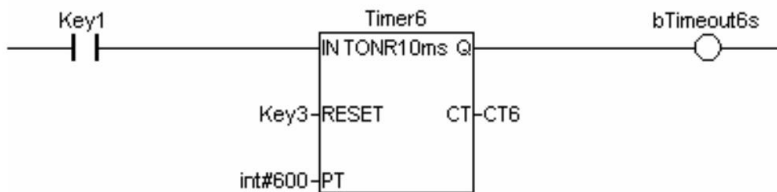
**or**



### **LD Language**



**or**



### **IL Language**

(\* TONR1 is a declared instance of TONR100ms function block \*)

Op1: CAL TONR1 (IN,RESET,PT)

LD TONR1.Q

ST Q

LD TONR1.CT

ST CT

**OR**

(\* TONR1 is a declared instance of TONR10ms function block \*)

Op1: CAL TONR1 (IN,RESET,PT)

LD TONR1.Q

ST Q

LD TONR1.CT

ST CT

**See also**

TOF100ms TOF10ms TON100ms TON10ms

## ***Logic Modules***

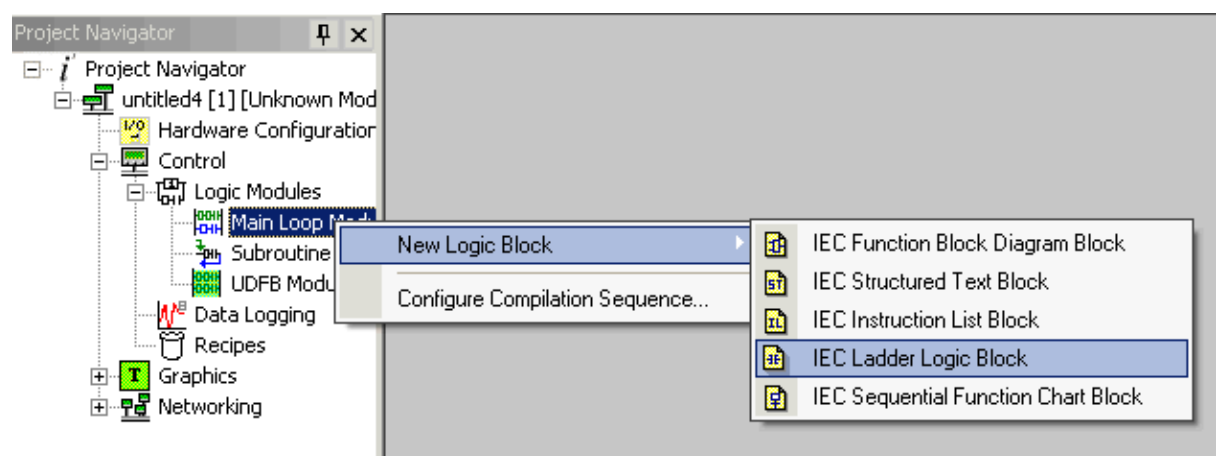
## IEC Modules

### Main loop modules for IEC programs

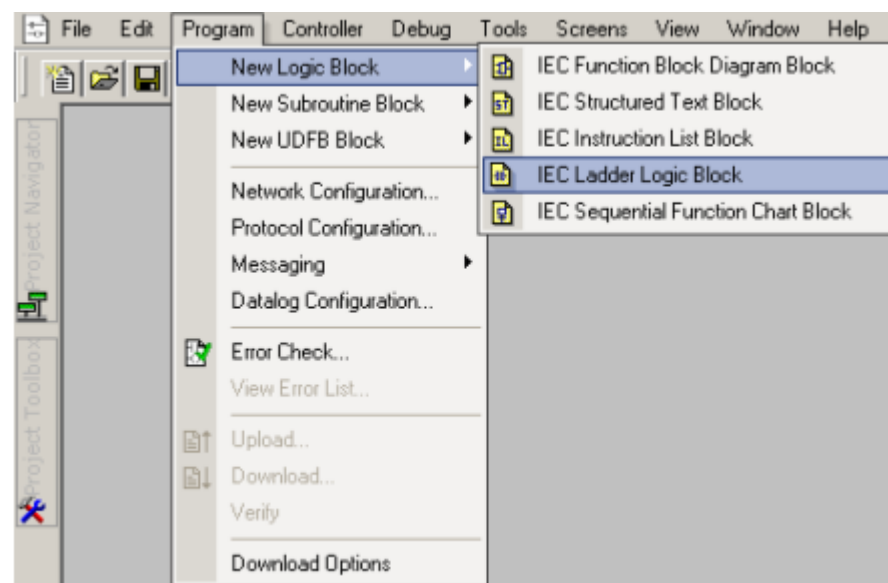
[Subroutine Modules](#) | [UDFB modules](#)

These blocks are executed once on each scan of the PLC in the order in which they are defined. User can configure the compilation sequence of multiple blocks in main loop modules area of an IEC Program by accessing the “IEC Modules Compilation Sequence Configuration” Dialog.

The programming language for main loop modules can be selected using right click on the Main Loop Modules node under Logic Modules from the program node of the Project Navigator.



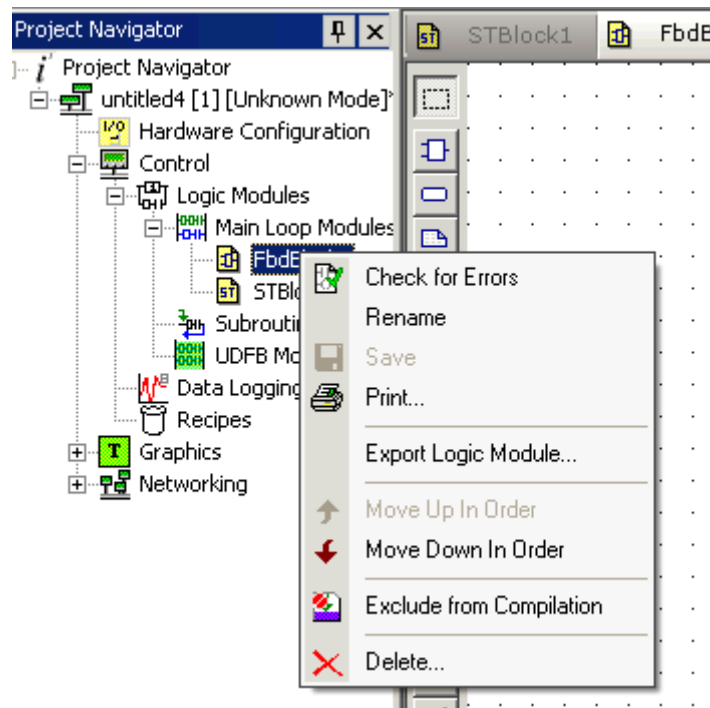
The programming language can also be selected by using the Program Menu | New Logic Block or through IEC Editor logic modules toolbar.



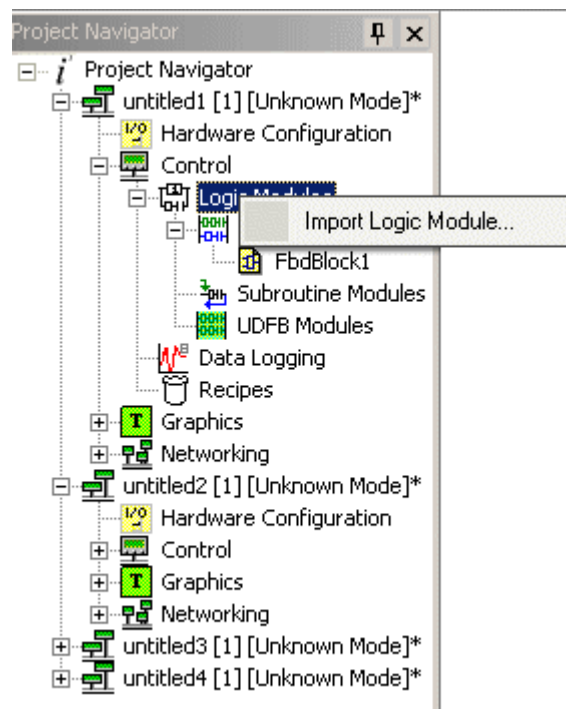
Note: SFC programs may only be created as Logic Blocks.

### ***Right Click Options***

The right click option allows the user to check for errors, rename, save, print, export logic module, Move Up In Order, Move Down In Order, Include/Exclude Compilation or delete the modules.

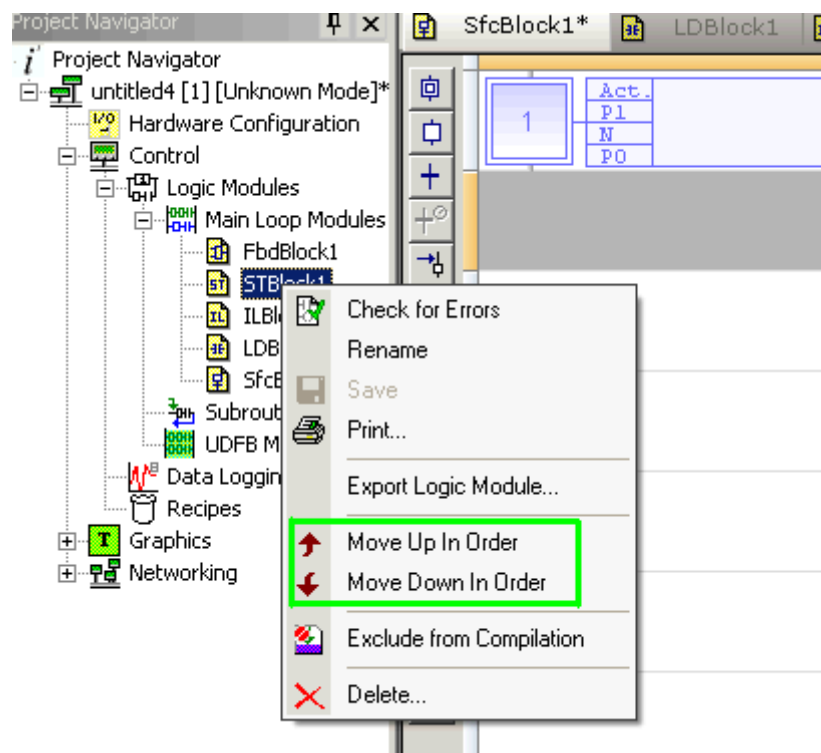
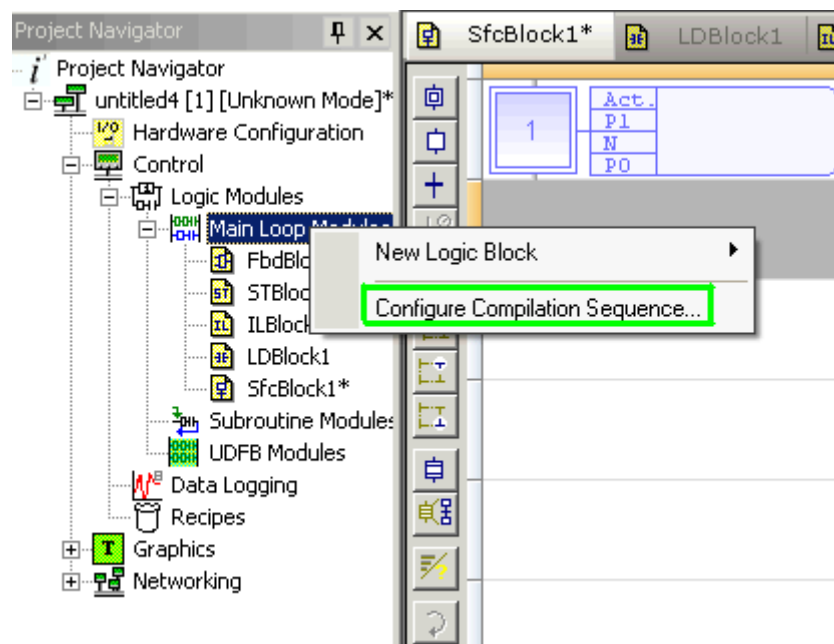


### ***Import Logic Module Right Click***

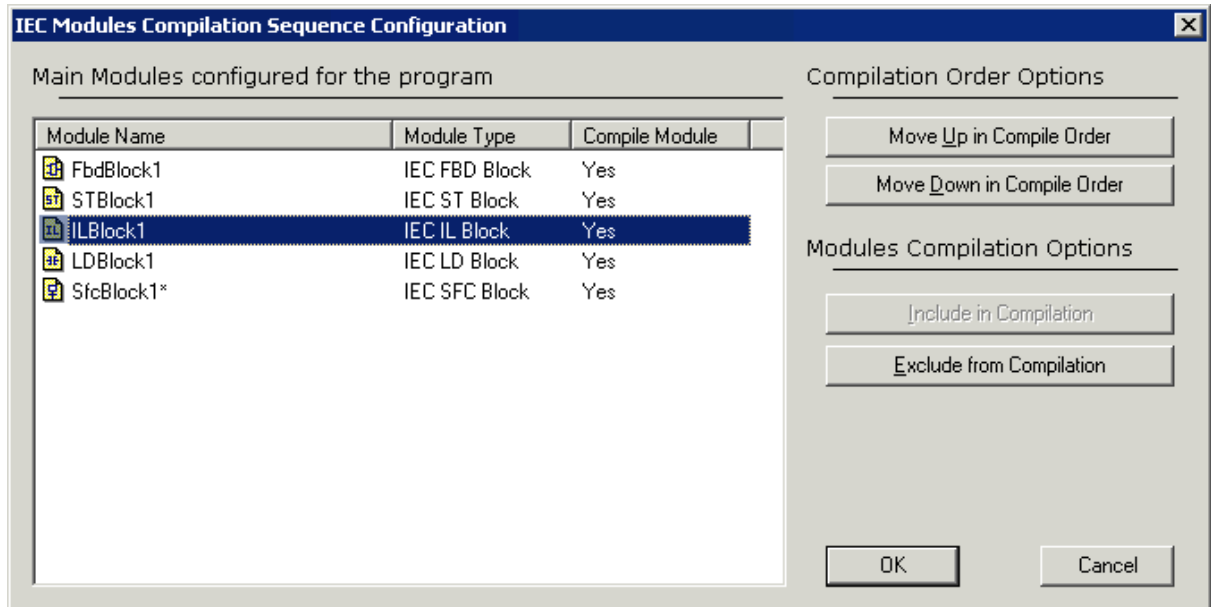


## IEC Modules Compilation Sequence Configuration

"Configure Compilation Sequence" can be accessed either by right clicking on the Main Loop Modules or right clicking on individual modules as shown below:



Selecting "Configure Compilation Sequence" gets the user to the below displayed window:



User can change the compilation order of the programs by using either Move Up in compile order or Move Down in Compile order buttons.

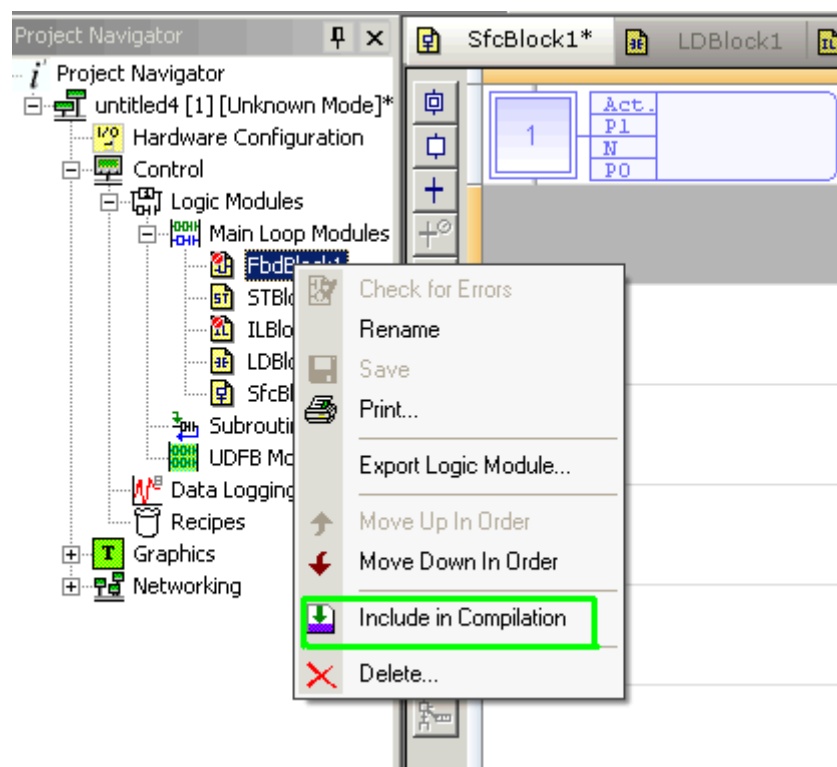
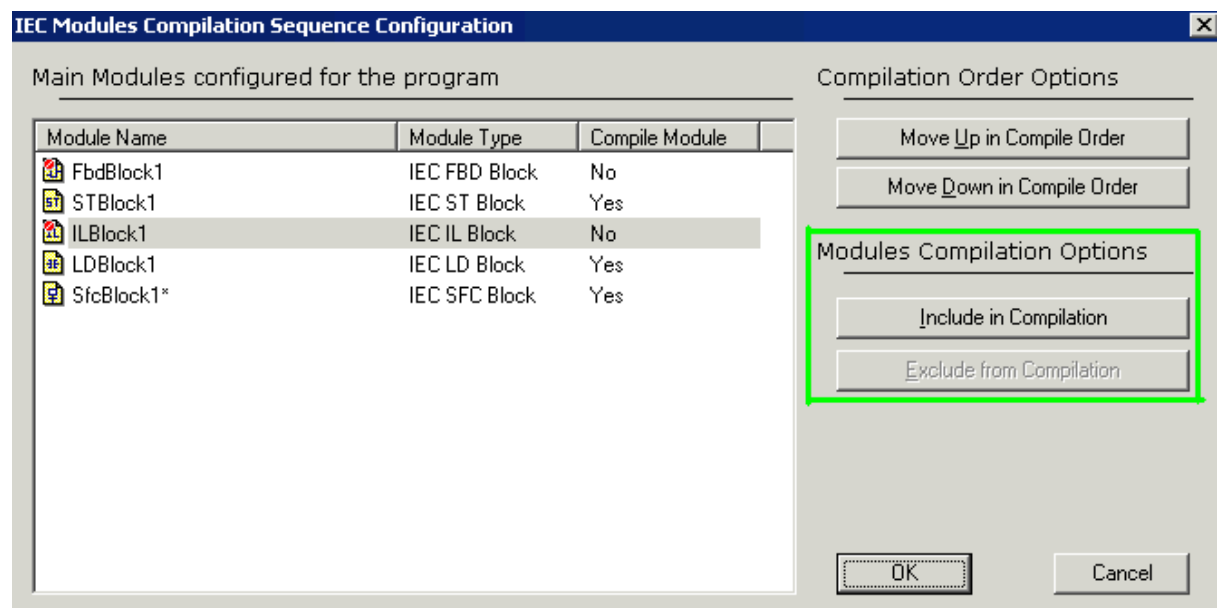
Move Up in Compile Order :- Moves the selected program one step up the compilation order.

Move Down in Compile Order :- Moves the selected program one step down the compilation order.

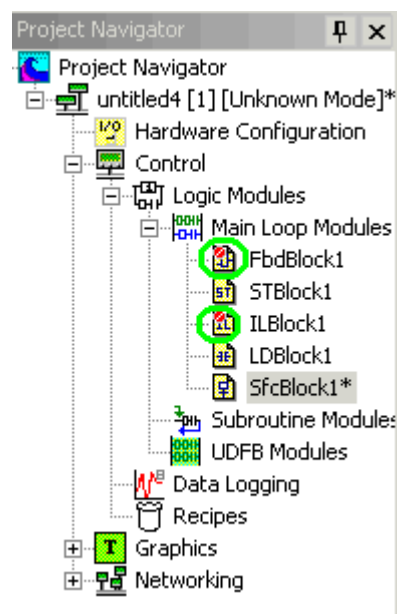
Note:- SFC block compilation order cannot be changed and they always remain last in the compilation order. But the compilation order of multiple SFC blocks can be changed within themselves.

User can also include/exclude a logic module in the main loop modules section from compilation sequence by using Include in Compilation / Exclude from Compilation options. This option can be selected either through IEC Modules Compilation Sequence Configuration window or by right clicking on any modules. These options are as shown below:





Any Module which is excluded from Compilation will have a **RED** mark indicated on the module as shown below:

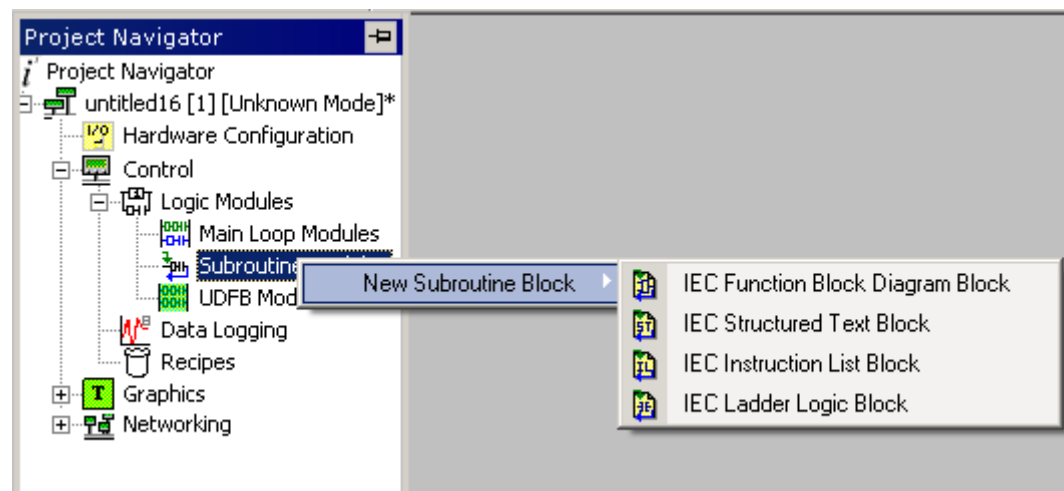


## Subroutine modules for IEC programs

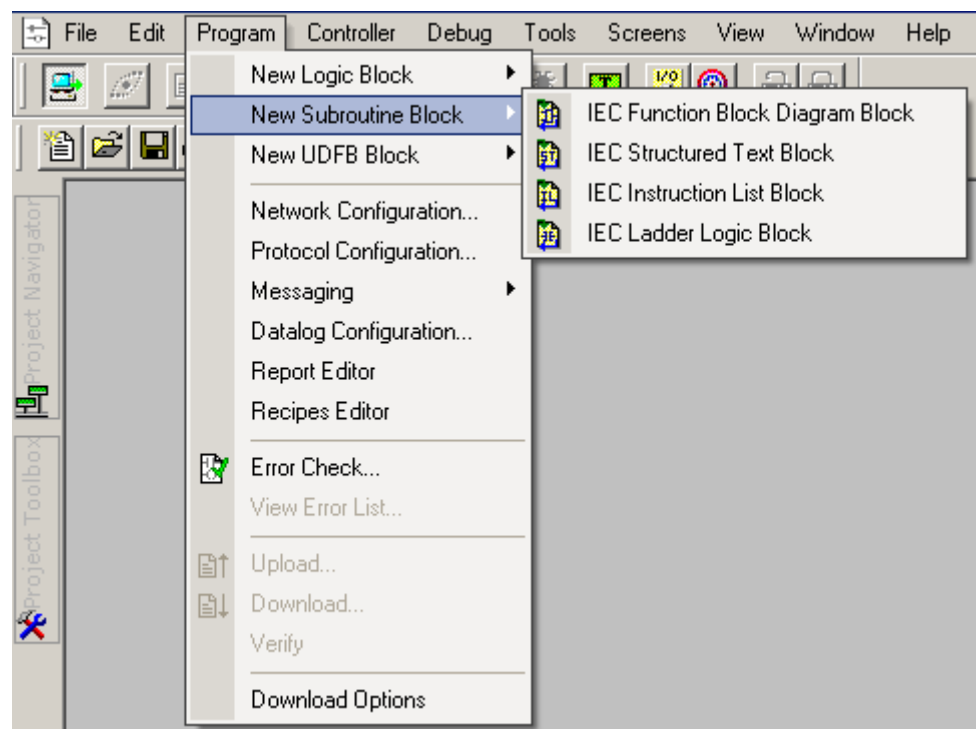
Main loop modules | UDFB modules

These blocks are callable from all the other block types. Subroutine modules can have private and local variables which will be allocated single storage. Hence calling the block from two different places will operate on the same private and local variables.

The programming language for subroutine modules can be selected using right click on the Subroutine Modules node under Logic Modules from the program node of the Project Navigator.



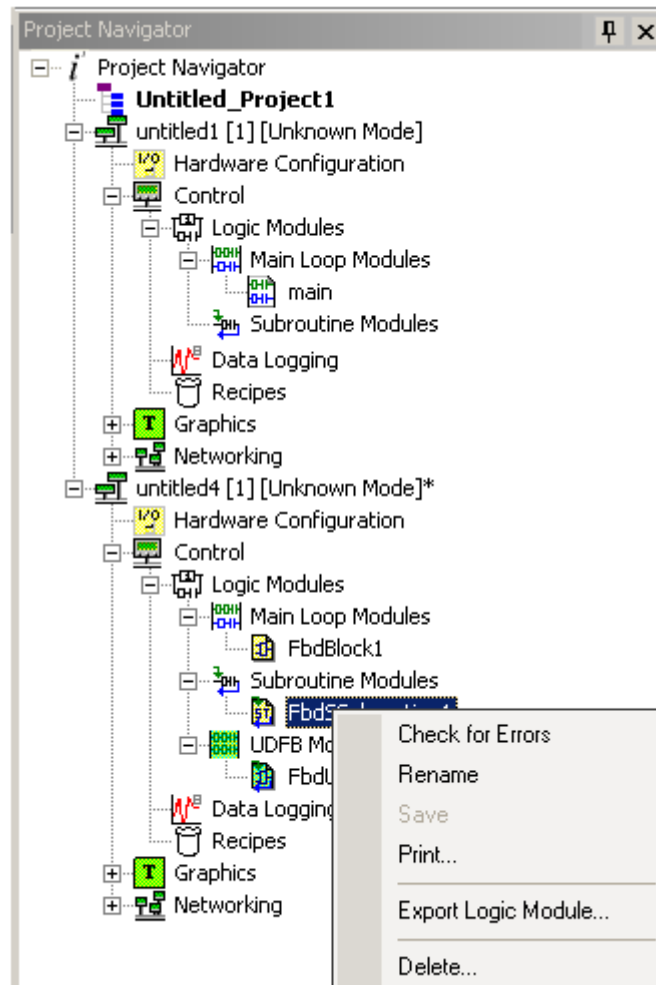
The programming language can also be selected by using the **Program Menu | New Subroutine Block** or through IEC Editor logic modules toolbar.



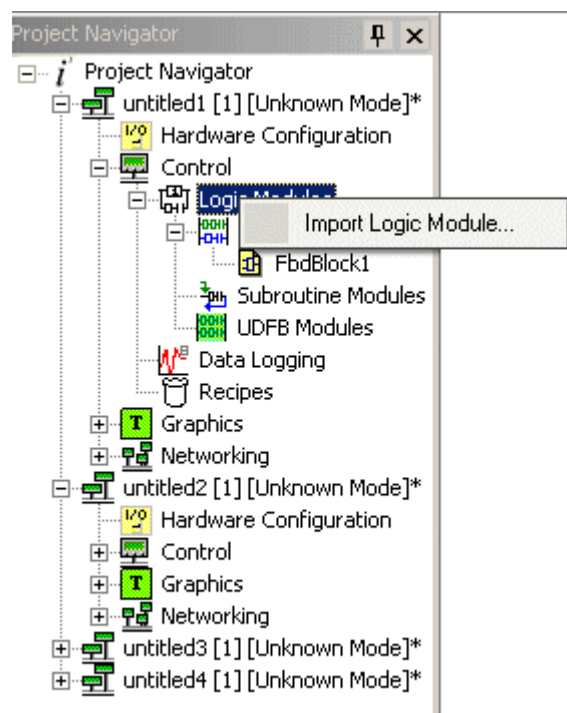
When a subroutine module is created, the same gets listed in the project toolbox under group 'Project'. These modules can then be dragged and dropped as other function blocks.

### ***Right Click Options***

The right click option allows the user to check for errors, rename, save, print, export logic module or delete the modules.



### ***Import Logic Module Right Click***

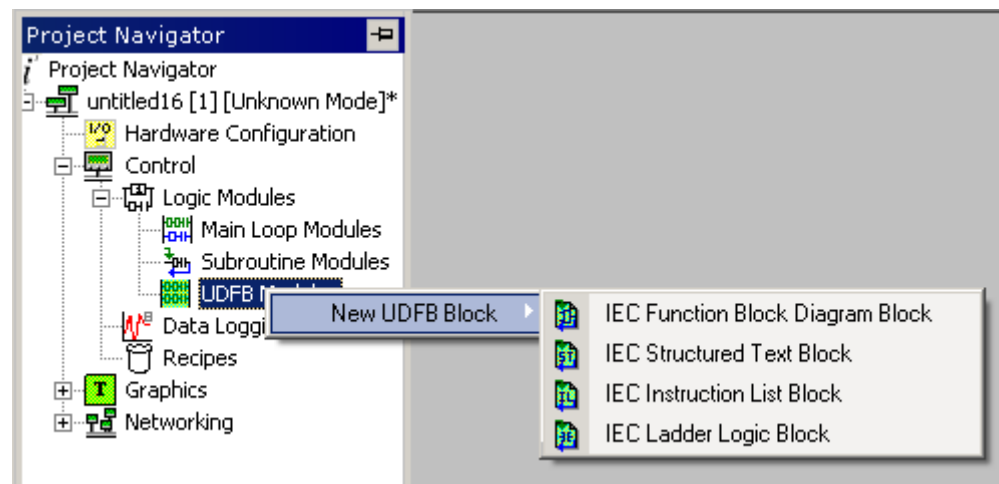


## UDFB modules for IEC programs

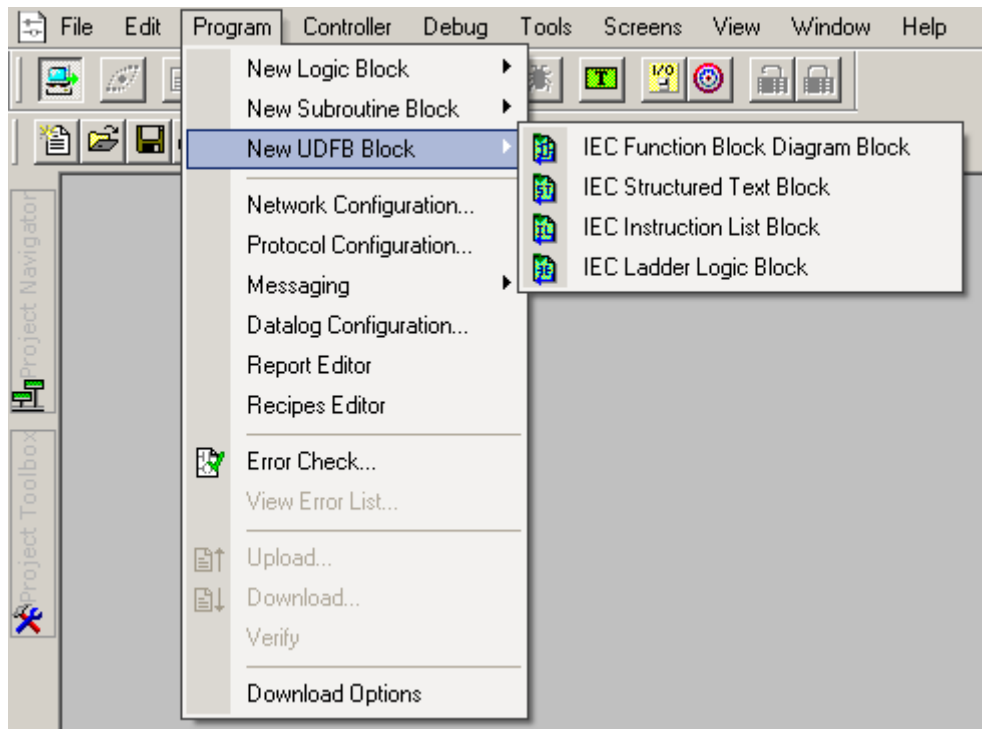
[Main loop modules](#) | [Subroutine modules](#)

These blocks are callable from all the other block types. UDFB modules can have private and local variables which will be allocated with unique storage. Hence calling the blocks from two different places will operate on different private and local variables.

The programming language for UDFB modules can be selected using right click on the UDFB Modules node under Logic Modules from the program node of the Project Navigator.



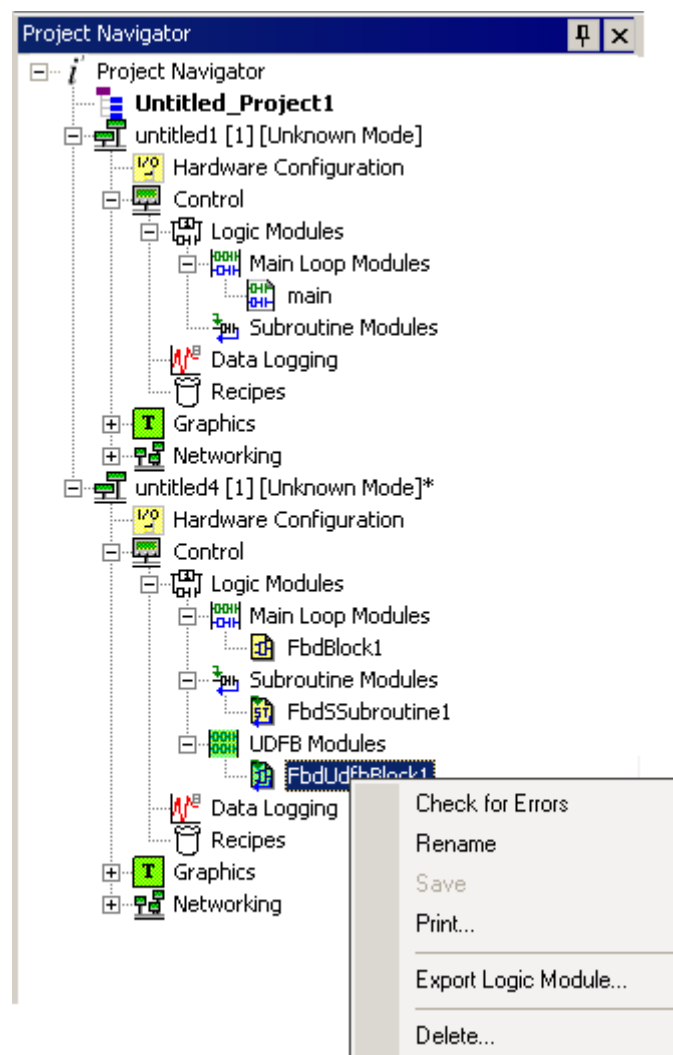
The programming language can also be selected by using the Program Menu | New UDFB Block or through IEC Editor logic modules toolbar.



When a UDFB module is created, the same gets listed in the project toolbox under group 'Project'. These modules can then be dragged and dropped as other function blocks.

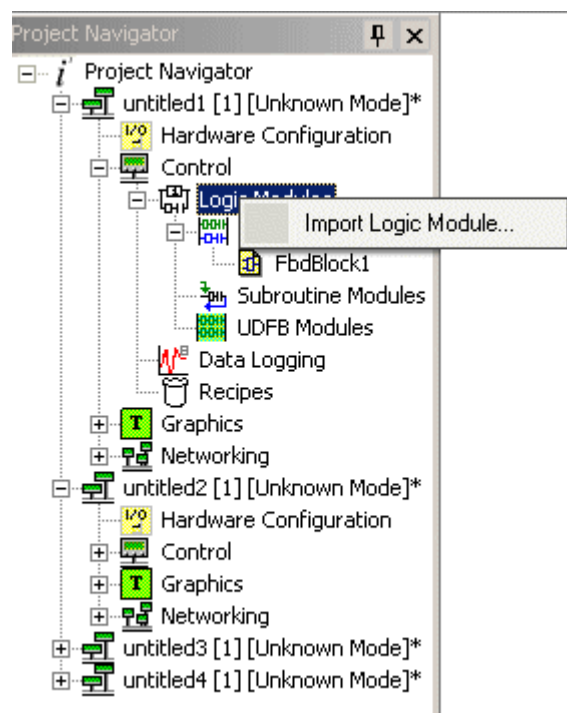
### ***Right Click Options***

The right click option allows the user to check for errors, rename, save, print, export logic module or delete the modules.



***Import Logic Module Right Click***

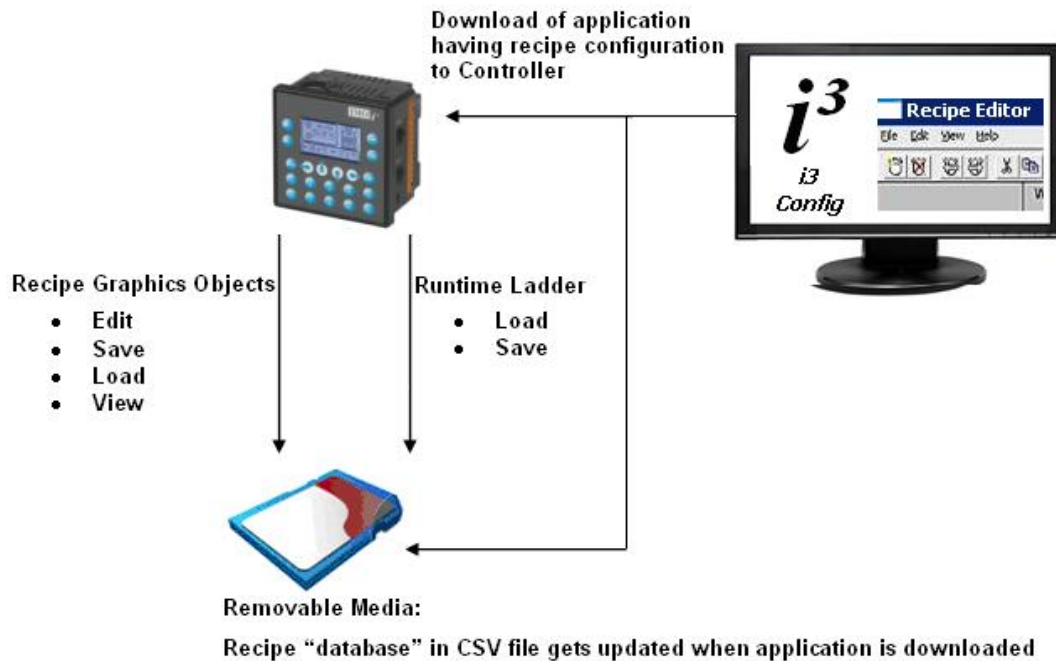




## Recipes

Recipes Ladder Elements | Recipes Graphics Object

### Overview



Recipes allow the user to send or update multiple registers simultaneously. For example, it may be desired to run a motor at two different settings for two different applications.

- Speed of 1000RPM, minimum frequency of 500Hz, acceleration rate of 1000 s/100Hz and deceleration rate of 2000 s/100Hz.
- Speed of 500RPM minimum frequency of 400Hz, acceleration rate of 500- s/100Hz and deceleration rate of 1500 s/100Hz.

Recipes enable the user to change all the fields (four in this example) at the same time without editing each individual field.

A maximum of 250 recipes and 1024 product can be supported. The maximum number of ingredients that can be supported across all recipes are 250. Recipe space is limited by the size of Removable Media put in the controller. The recipe function will require presence of a removable media card to store the recipe data.

If a program having recipe configured is exported, **i3 Configurator** will create 2 files namely \*.pgm and \*.csv. User should copy both these files to Removable Media for loading in the device.

**Also See:** Recipes Settings in download options.

## Creating a Recipe

1. Open the **Create New Recipe Database** dialog by selecting **Program | Recipes Editor** from the Main Menu.
2. Enter the **Recipe Name**, **File Name**, **Number of Products** and **Number of Ingredients**.



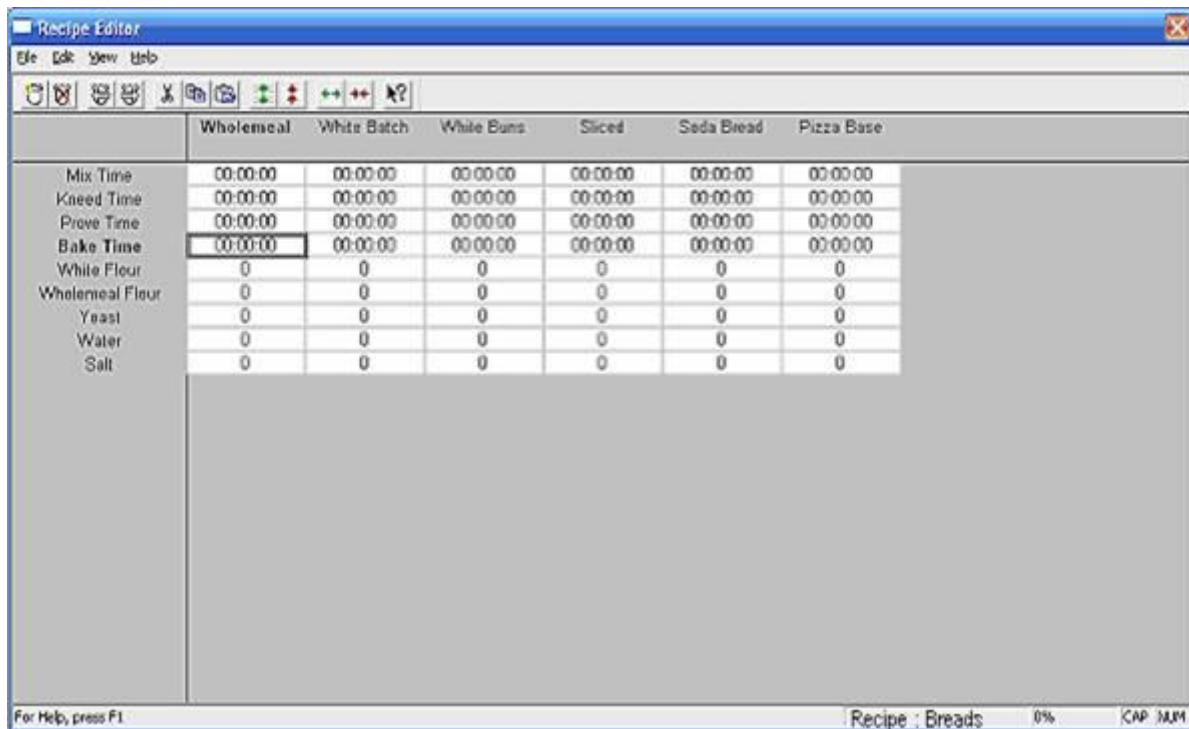
When a new recipe is created the following data is required:

Create Recipe Settings	
<b>Recipe Name</b>	Used to label the recipe in the editor. It is also used in the associated graphic elements to identify the required recipe. Allowed characters are a-z, A-Z, 0-9 and _.
<b>Filename</b>	Indicates the filename (plus path if required) where the data for the recipe is stored on the removable media card.
<b>Number of Products</b>	Enter up to 8 characters for the <b>File Name</b> . File name must be in 8.3 format with .CSV extension. Initial number of products to be allocated for the recipe. "The number of products/ingredients" may be easily changed during editing by inserting or deleting columns/rows from the recipe database.
<b>Number of Ingredients</b>	Initial number of ingredients to be allocated for the recipe. "The number of products/ingredients" may be easily changed during editing by inserting or deleting columns/rows from the recipe database.

Clicking on **OK** opens the **Recipe Editor** dialog.

### Recipe Editor

The Recipe editor dialog is as follows:



Different products are listed across the top of the spreadsheet. Clicking on the product name will allow editing of the product name. This is the string which will be used to select a recipe product for loading/saving of the control registers in the [i3](#).

Different ingredients to be loaded are listed in the left most column. Clicking on the setting name will allow editing of ingredient properties, which include name to be displayed, the register(s) in which the data is placed and the format in which the data for that ingredient will be displayed.

The data for the products is contained in the central ‘spreadsheet’ area. This is the data which is stored on the removable media card.

## Recipe Editor

### Toolbar Buttons

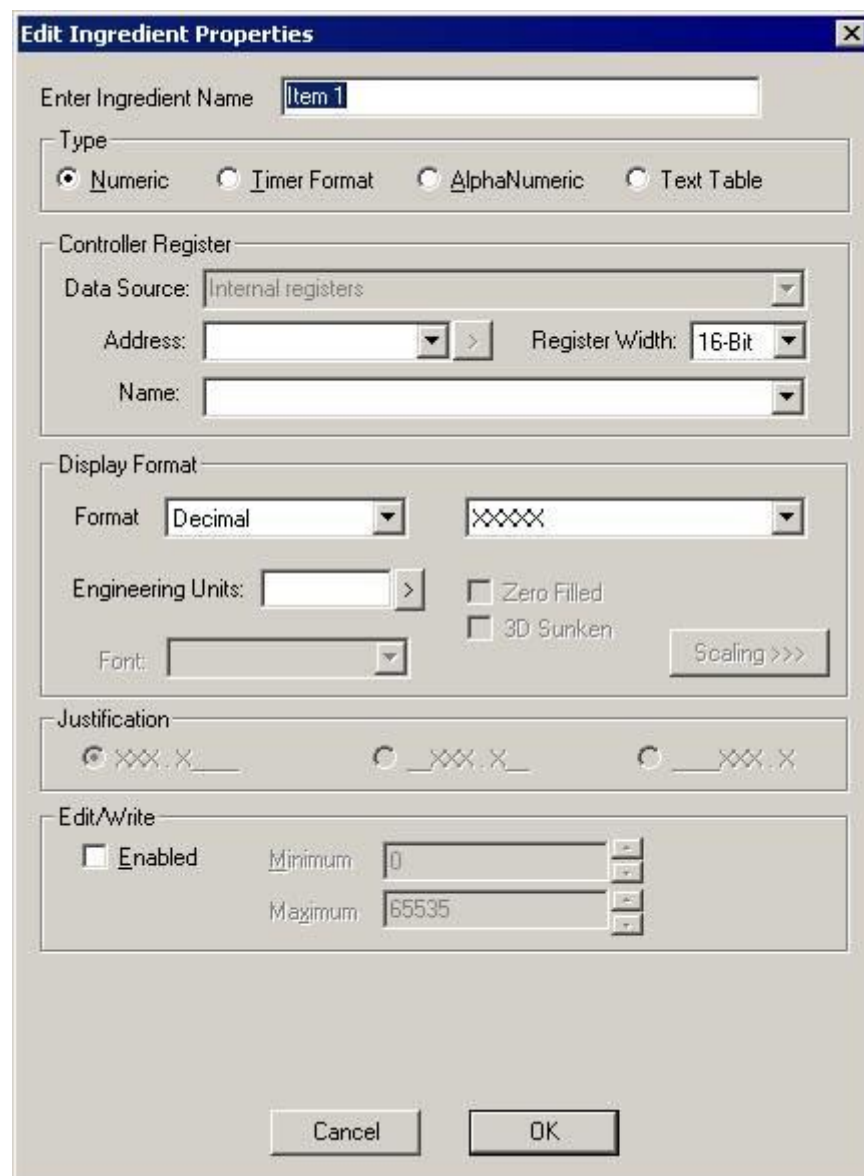
	Create a new Recipe
	Delete: Delete currently selected recipe. User is prompted for confirmation
	Previous/Next Recipe: Move through the recipes in the program. This is augmented by the names of all recipes being available for selection in the Tools menu.
	Cut/Copy/Paste: Standard windows functionality. Data is saved to the clipboard in text format allowing easy transfer to other applications.
	Insert/Delete Ingredient: Delete only after confirmation
	Insert/Delete Ingredient: Delete only after confirmation

## Menu Bar



## Editing Ingredient Properties

An ingredient, or field, in the recipe corresponds to a value which will be loaded to a specified [i3](#) register.

A screenshot of the 'Edit Ingredient Properties' dialog box. The dialog has a title bar with a close button. It contains several sections: 'Enter Ingredient Name' with a text field containing 'Item 1'; 'Type' with radio buttons for 'Numeric' (selected), 'Timer Format', 'AlphaNumeric', and 'Text Table'; 'Controller Register' with a 'Data Source' dropdown set to 'Internal registers', an 'Address' dropdown, a 'Register Width' dropdown set to '16-Bit', and a 'Name' dropdown; 'Display Format' with a 'Format' dropdown set to 'Decimal', a pattern dropdown set to 'XXXX', 'Engineering Units' dropdown, a 'Font' dropdown, checkboxes for 'Zero Filled' and '3D Sunken', and a 'Scaling >>>' button; 'Justification' with three radio buttons for different alignment patterns; and 'Edit/Write' with an 'Enabled' checkbox, 'Minimum' and 'Maximum' value fields (0 and 65535 respectively), and increment/decrement buttons. At the bottom are 'Cancel' and 'OK' buttons.

Each ingredient has the following properties:-

Editing Ingredient Settings
-----------------------------

<b>Ingredient Name</b>	The ingredient name which will appear in the graphics object for editing the ingredient can be provided.
------------------------	--

**Note:** *i*<sup>3</sup> can use up to 32-bytes to store the name of the selected product.

**Type** Select the data type of the ingredient.

**Controller Register** Select the *i*<sup>3</sup> register where the data will be loaded. (Register name & Address)

Select the data format, justification, decimal position etc.

**Numeric**

The 'Numeric' dialog box is divided into two sections. The top section, 'Display Format', contains a 'Format' dropdown menu set to 'Decimal', a 'Precision' dropdown menu set to 'X', an 'Engineering Units' text box with a right arrow, a 'Font' dropdown menu, and two checkboxes for 'Zero Filled' and '3D Sunken'. A 'Scaling >>>' button is located to the right. The bottom section, 'Justification', contains three radio buttons with corresponding patterns: 'XXX.X', '\_.XXX.X', and '\_.XXX.X'.

**Timer Format**

The 'Timer Format' dialog box has a 'Display Format' section with a 'Timer Format' dropdown menu set to 'HH:MM:SS' and a 'TimeBase' dropdown menu set to '1 s'. The 'Justification' section below contains three radio buttons with patterns: 'XXX.X', '\_.XXX.X', and '\_.XXX.X'.

**Display  
Format**

**AlphaNumeric**

The 'AlphaNumeric' dialog box has a 'Display Format' section with a 'Number of Characters' spinner box set to '0'. The 'Justification' section below contains three radio buttons with patterns: 'XXX.X', '\_.XXX.X', and '\_.XXX.X'.

**Note:** The alphanumeric data cannot have comma in it. If comma is put, it will be replaced with space.

**Text Table**

The 'Text Table' dialog box has a 'Display Format' section with a 'Number of Characters' spinner box set to '5', a 'Table ID' text box set to '1', and an 'Edit Table' button. The 'Justification' section below contains three radio buttons with patterns: 'XXX.X', '\_.XXX.X', and '\_.XXX.X'.

## Edit/Write

Enable the check box if the value of the ingredient can be edited in the *i<sup>3</sup>* and provide the maximum and minimum values which are used when entering the data in *i<sup>3</sup>*.

## Editing Recipe Data

Double click on the field to be edited. If the data is a bit status it will toggle, otherwise the **Edit Field Contents** dialog box will appear. Enter the new data for the field and press enter or click **OK**.

A dialog box titled "Enter Field Contents" with a close button (X) in the top right corner. It contains a text input field labeled "Enter Value" and two buttons at the bottom: "OK" and "Cancel".

## Renaming Products

Double click on the record name. When the **Enter Product Name** dialog box is displayed enter the new record name, and click **OK** or press Enter.

**Note:** *i<sup>3</sup>* can use up to 32-bytes to store the name of the selected product.

A dialog box titled "Enter Product Name" with a close button (X) in the top right corner. It contains a text input field labeled "Enter Name" with the text "Product 1" inside. Below the field are two buttons: "OK" and "Cancel".

## Auto Allocate Ingredient Register

A dialog box titled "Auto Allocate Ingredient Registers" with a close button (X) in the top right corner. It contains four fields: "Start Ingredient" with a dropdown menu showing "Item 1", "End Ingredient" with a dropdown menu showing "Item 4", "Registers Required" with a text input field containing "4", and "Allocate from" with an empty text input field. To the right of the "Registers Required" field is a yellow label that says "16-BIT". At the bottom are two buttons: "OK" and "Cancel".

## Auto Allocate Ingredient Register

### Start Ingredient

Select Start ingredient name for a range from this dropdown.

### End Ingredient

Select End ingredient name for a range from this dropdown.

<b>Register Required</b>	This displays the number of registers that will be consumed for items/ingredients selected from start to end ingredients, which will be equal to number register used for auto allocation. This field can not be edited.
<b>Allocate from</b>	Starting address of the register for auto allocation.

## Editing Current Recipe

This opens the Recipes storage details dialog which allows changes to the recipe name and file name to be made.

## Configuring Product Register

This register is used for storing the selected product of the recipe. The Enable Product Register checkbox must be checked before Product Register can be used.

**Note:** To view selected product name or index in the View recipe graphics object, this option must be configured.

## Product Register Settings

Selecting this option will store the Index of the most recently selected product in the register provided here.

The product can be selected from Load Recipe ladder blocks or **Load Recipe** graphics objects.

**Index**



The image shows a dialog box titled "Product Register". It contains a checked checkbox labeled "Enable Product Register". Below this are two radio buttons: "Index" (which is selected) and "Name". At the bottom, there are two empty text input fields. A yellow label "16-BIT" is positioned between the two input fields. At the bottom right of the dialog are "OK" and "Cancel" buttons.

**Note:** *i3* uses 16-bits to store index of the selected product.



Selecting this option will store the Name of the most recently selected product in the register provided here.

The product can be selected from Load Recipe ladder blocks or **Load Recipe** graphics objects.

**Name**

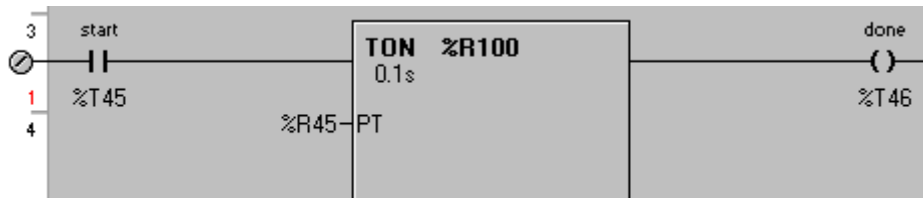
The image shows a 'Product Register' dialog box with a blue title bar and a close button. Inside, there is a checked checkbox labeled 'Enable Product Register'. Below it are two radio buttons: 'Index' (unselected) and 'Name' (selected). Under the 'Name' radio button, there is a text input field with a dropdown arrow. To the right of this field is another empty text input field with a dropdown arrow. At the bottom right are 'OK' and 'Cancel' buttons.

**Note:** *i<sup>3</sup>* can use up to 32-bytes to store the name of the selected product.

## Using Setpoints

### Extended %R Registers

Setpoints allow registers to be initialized to a known value after a download. The following is an example of using setpoints:

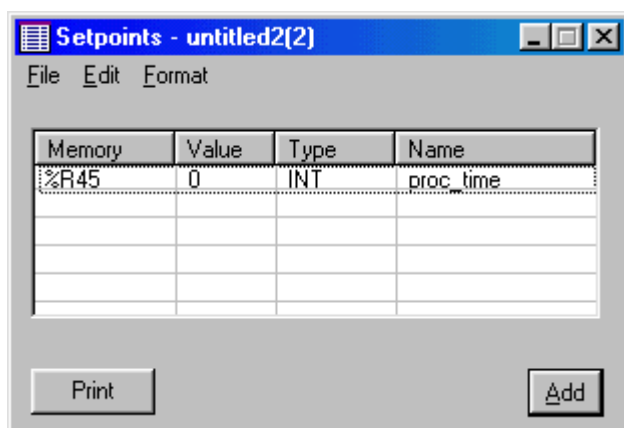


A timer with a variable setpoint (**%R45**) is used to control a process. The setpoint should be approximately 470 milliseconds but it requires some tuning to determine the exact value. If you just download this program **%R45** has a unknown value because it has not been initialized.

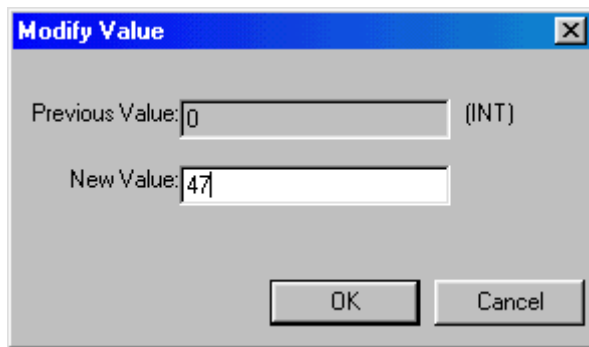
Right click on the timer and choose **Add to Setpoints**:



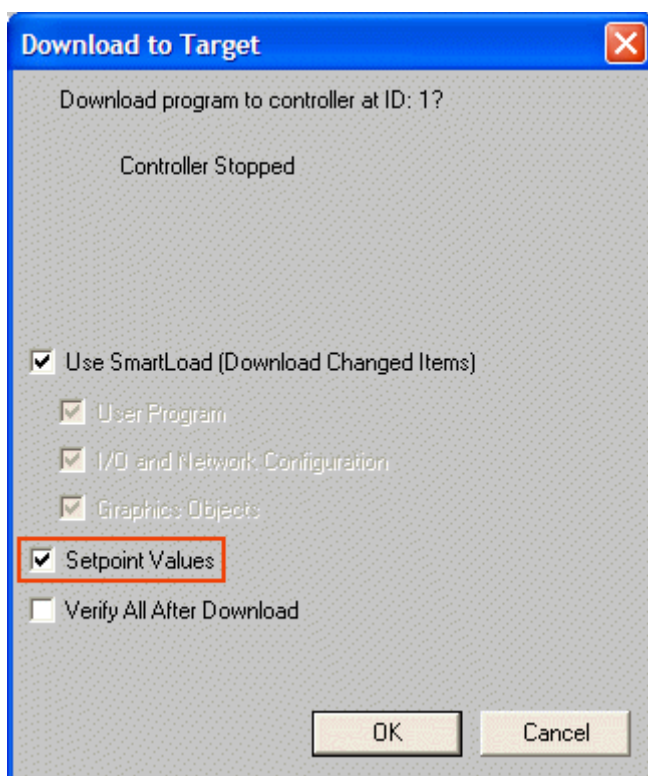
Because **%R45** is a variable input it will be added to the setpoint list:



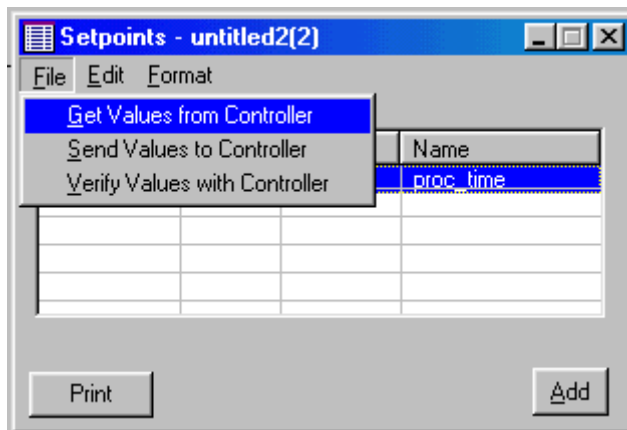
Now one can define the initial value for this setpoint, **47** for 470 milliseconds:



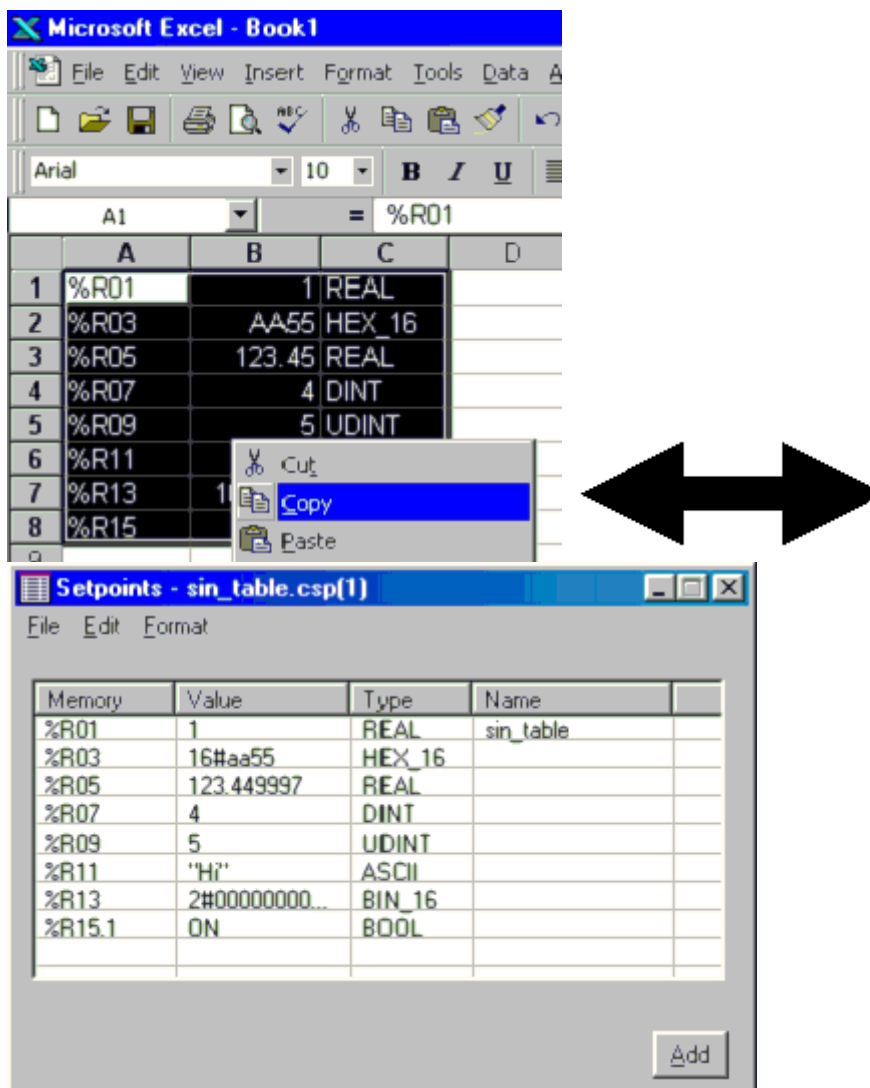
When downloading the program select the Set Point Values in the list of items to download. This will load the register **%R45** with a 47 when the download takes place:



After tuning the timer to the actual process one finds that the optimal value for the timer is 440 milliseconds. This value can be manual recorded in the setpoint table, or all the setpoints can be uploaded from the controller into *i<sup>3</sup> Configurator*'s memory for storage when the program is saved:



## Editing the Setpoints



Setpoints can be cut, copied, pasted and deleted. Cut, copy and paste works with *i<sup>3</sup> Configurator* or other Windows programs such as Microsoft Excel or Word.

## Formatting Setpoints

Setpoints can be formatted to any type that is required. These type include:

BOOLEAN, BINARY-16, HEX-16, INT, UINT, DINT, UDINT, ASCII, and REAL

### Transferring and Verifying Values with the Controller

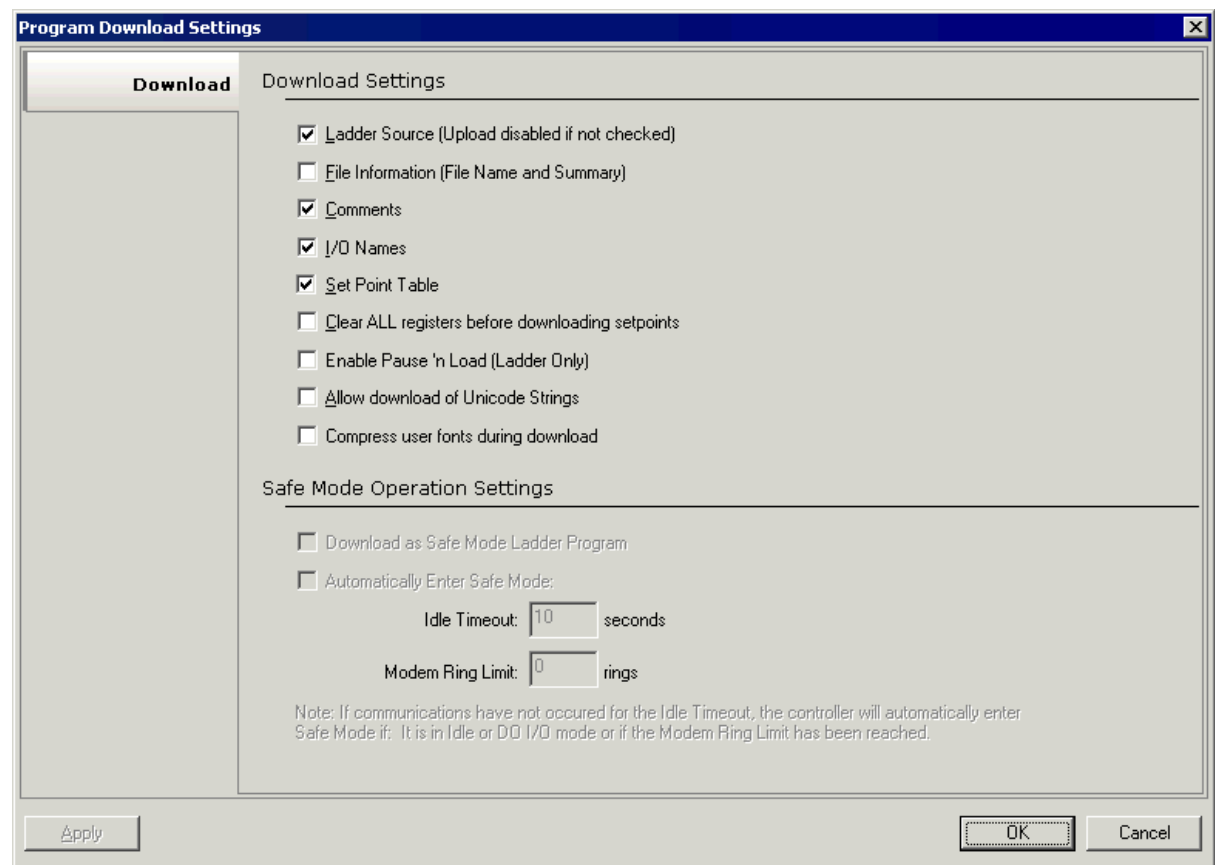
The values from the setpoint table can be sent to the controller, obtained from the controller, or verified with the controller using the **FILE** menu of the setpoint dialog.

### Setpoint Tables, Setpoint Values, Uploading and Downloading

A **Setpoint Table** is a list of registers and data types. **Setpoint Values** are the numeric values that are associated with the setpoint.

When you download a program and select to download the setpoint values, *i<sup>3</sup> Configurator* sends commands to change the values of the registers in the controller as the user has defined in the setpoint table (this is the same as selecting Send Values to Controller from the setpoint menu). The controller has no record of what registers were defined as setpoints. This information is stored in the *i<sup>3</sup> Configurator* ladder program and is saved to disk with the program.

*i<sup>3</sup> Configurator* can download this table and the original values stored in the table for archival purposes. By selecting the **Program -> Download Options** menu item the following dialog will appear.



By checking the Download Setpoint Table option, the table is downloaded to the controller's nonvolatile storage when the program is downloaded.

When a program is uploaded if this option was checked, the uploaded file will contain the setpoint table and the original values that were entered into the table. Because the ladder program or text screens can modify these values in the controller registers, the values in this uploaded table may not represent the values in the controller. By selecting **Get Values from Controller** in the setpoint **File** menu, the values from the controller will be uploaded and will replace the original values from the upload.

### **Printing the Setpoints**


The setpoint can be printed in a tabular format when any part of the ladder program is being printed by selecting the setpoints item on the print setup.

The setpoints can also be printed using the **Print** button found on the setpoint dialog.

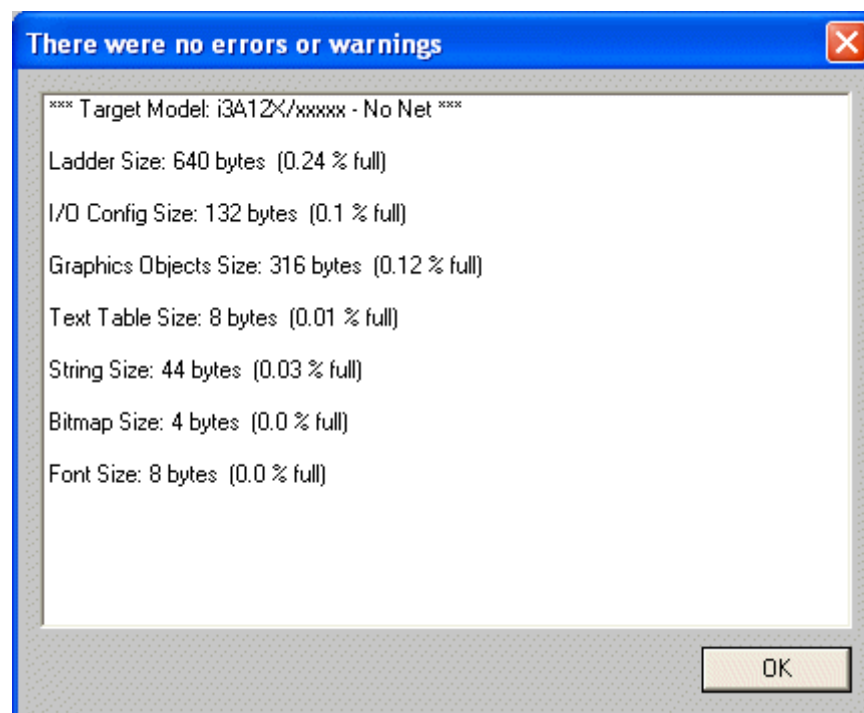
## How to Check a Program for Errors

### Error and Warning List

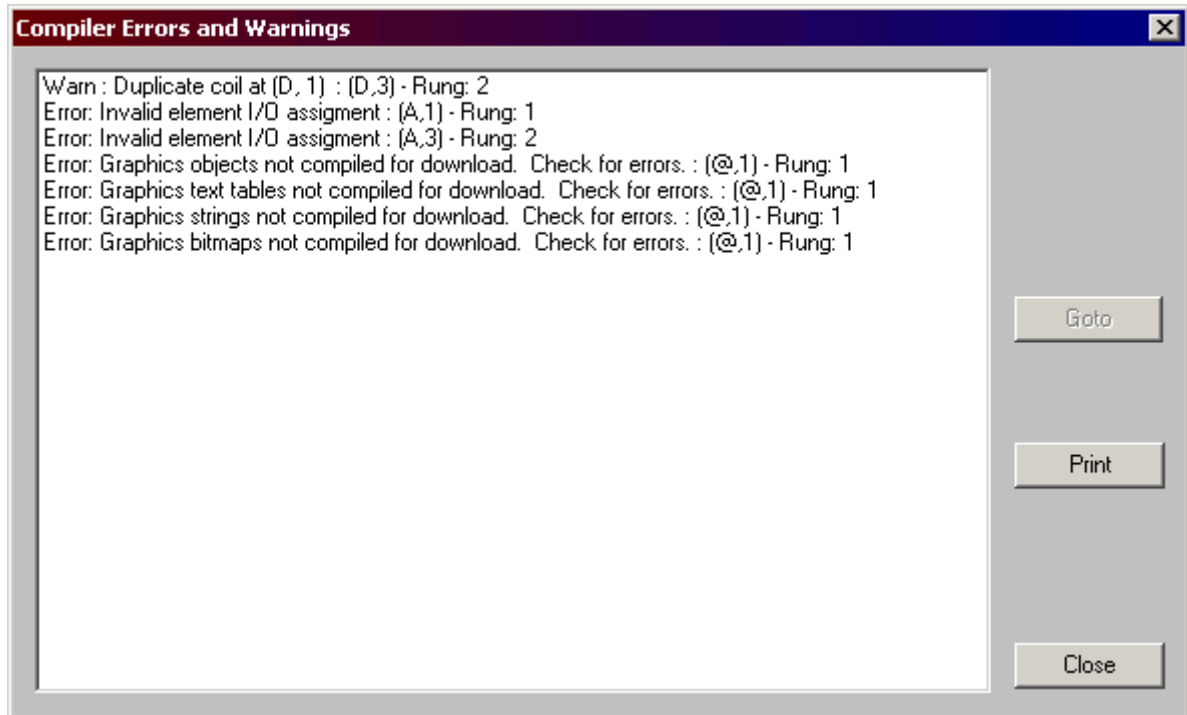
Ladder Programs are automatically checked for syntactical errors before they are downloaded to the controller. The Graphics portion is checked first and then the Ladder portion is checked.

Manually check a program for errors by selecting **Program | Error Check..** from the Main Menu or selecting the Error Check tool .

If no errors occur, then a message box appears:



Otherwise, the Error List is displayed



There are two kinds of listings, **ERRORS** and **WARNINGS**.

**ERRORS** are problems that prevent the program from running, such as unconnected elements. **ERRORS** must be corrected before the program can be downloaded.

**WARNINGS** are problems which can cause difficulty or unexpected operation of the controller, but are otherwise syntactically correct (i.e., using the same output coil at two different points in the program). This can be intentional. **WARNINGS** must be checked to see that they produce the desired results without unwanted side effects.

To move directly to the offending rung in the Ladder Code program, double click on the error, or single click on the error and click **GOTO**.

### Error and Warning List

The Error List uses the following Syntax:

**[Status]: [Description] : [Location]**

Where:

**[Status]** ERROR or WARN

**[Description]** gives a short description of the problem

**[Location]** gives the Row and Column location of the offending element.

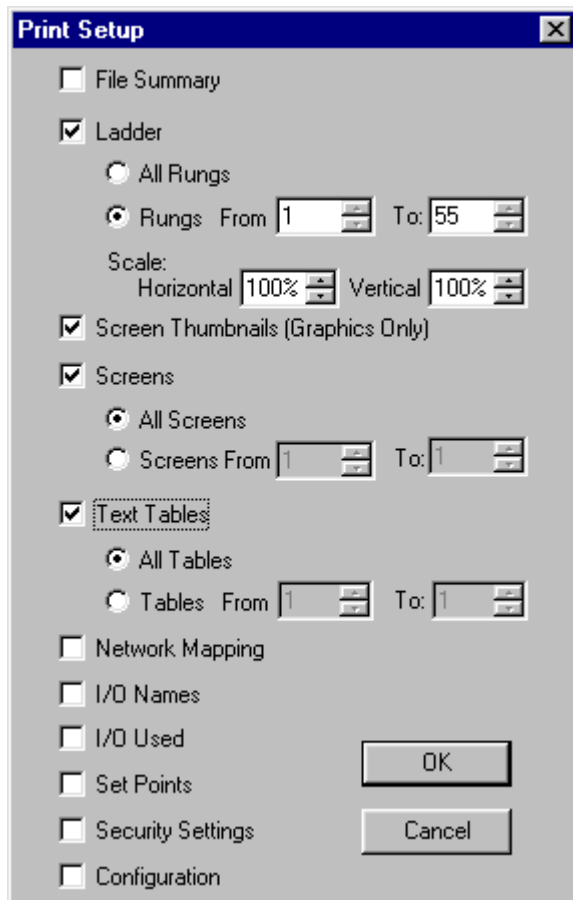
Example: **Error : Output without an input : Main(A,5)**

In some cases, the description gives a location in the line of the offending element.



## Print Setup Dialog

This dialog allows the selection of information included on a print-out.



**File Summary** - This prints the project name, author, full file name and location, date created, date last modified, *i<sup>3</sup> Configurator* version, and any notes provided.

**Ladder** - This allows a range of ladder rungs to be printed. The size of the ladder program can be scaled from 20 to 200 percent. This controls the size of the ladder printout. Note that the horizontal and vertical size can be adjusted independently. The text in the ladder program is scaled to match the vertical size only.

**Screen Thumbnails** - This allows smaller thumbnails of the graphics screens to be printed for reference. Typically 28 screens fit on an 8.5 x 11 inch piece of paper using a 4 x 7 grid of screens.

**Screens** - This allows a range of text or graphics screens to be printed with data field information.

**Text Tables** - This allows a range of text tables to be printed.

**Network Mapping** - This prints the network configuration for the program.

**I/O Names** - This prints a list of registers and their assigned names.

**I/O Used** - This prints a list of registers used in this ladder program, their assigned name, and how they are used in the program.

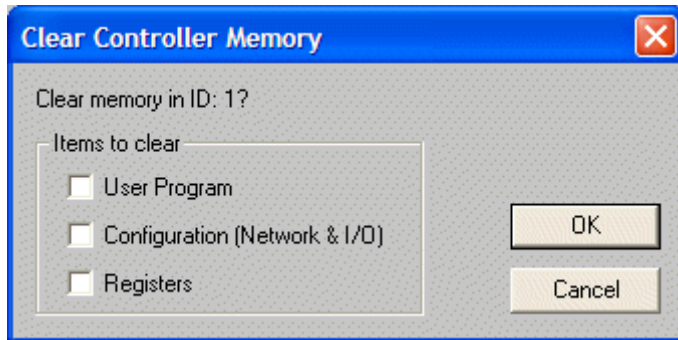
**Set Points** - This allows the table of setpoint to be printed.

**Security Settings** - This allows all the security information to be printed. If this user is logged on with an administrator password, the actual passwords are printed, otherwise only the names and privilege levels are printed.

**Configuration** - This prints the CPU and I/O configuration for this program.

## ***Clearing the Controller Memory***

To clear a controller's memory, first select the controller as the **target**, then select **Controller | Clear Memory** from the main menu. The Clear Memory dialog appears:



The network ID at the top of the box specifies which controller will have its memory cleared.

Select the memory areas to be cleared:

**User Program** -- Clears out the User Program memory area.

**Configuration (Network and I/O)** - Clears out the Configuration of the network and I/O areas.

**Registers** - Clears out all registers

IMO Precision Controls Limited  
1000 North Circular Road  
Staples Corner  
London NW2 7JP  
United Kingdom

Tel: +44 (0)20 8452 6444  
Fax: +44 (0)20 8450 2274  
Email: [imo@imopc.com](mailto:imo@imopc.com)  
Web: [www.imopc.com](http://www.imopc.com)

IMO Jeambrun Automation SAS  
Centre D'Affaires Rocroy  
30, Rue de Rocroy  
94100 Saint-Maur-Des-Fosses  
France

Tel: +8000 452 6444  
Fax: +8000 452 6445  
Email: [info@imopc.fr](mailto:info@imopc.fr)  
Web: [www.imopc.fr](http://www.imopc.fr)

IMO Automazione  
Via Ponte alle Mosse, 61  
50144 Firenze (FI)  
Italia

Tel: +39 800 783281  
Fax: +39 800 783282  
Email: [info@imopc.it](mailto:info@imopc.it)  
Web: [www.imopc.it](http://www.imopc.it)

IMO Canada  
18 Strathearn Avenue  
Unit 32- B-North,  
Brampton, ON L6T 4Y2  
Canada

Tel: +1 866 275 9688  
Fax: +1 905 799 0450  
Email: [imocanada@imopc.com](mailto:imocanada@imopc.com)  
Web: [www.imopc.com](http://www.imopc.com)